About This Book

This book is designed as a programmer's guide for the IBM SystemView Agent Version 1.4 program (hereafter referred to as SystemView Agent). The SystemView Agent provides the Simple Network Management Protocol (SNMP) agent distributed protocol interface (DPI). The SNMP DPI permits users to dynamically add, delete, or replace management variables in the local Management Information Base (MIB) without requiring you to recompile the SNMP agent.

Who Should Use This Book

You should read this book if you are an agent programmer who will use SystemView Agent. You should use this book if you are doing one or more of the following:

- Developing an SNMP subagent that uses the Distributed Protocol Interface (DPI) 2.0
- Migrating your SNMP DPI subagent to Version 2.0

Before reading this book, you should have a general understanding of network management, the operating system on which you are working, and the C programming language.

In addition, you may want to obtain the following documents:

- RFC1592 is the SNMP DPI 2.0 RFC.
- RFC1901 through RFC1910 are the SNMP Version 2 RFCs.

How to Use This Book

Read Introduction to SNMP Distributed Protocol Interface for information about the SNMP DPI routines supported by SystemView Agent and for information about running a DPI API application.

Read Subagent Programming Concepts for conceptual information about subagent programming. This section contains detailed information on request processing and contains specific programming recommendations.

Read Basic DPI API Functions for information on these DPI functions, including syntax and examples.

Read Transport-Related DPI API Functions for information on each of the DPI transport-related functions available. These functions try to hide any platform specific issues for the DPI subagent programmer so the subagent can be made as portable as possible. The information includes syntax and examples.

Read DPI Structures for information about each data structure used in the SNMP DPI API.

Read Character Set Selection for information about specifying the correct character set.

Read Constants, Values, Return Codes, and Include File for information about constants and values as they are defined in the snmp_dpi.h include file.

Read SNMP DPI API Version 1.1 Considerations for guidelines on programming with Version 1.1.

Read Migrating Your SNMP DPI Subagent to Version 2.0 for guidelines on programming with Version 2.0.

A DPI Subagent Example contains example code and comments that explain the example program.

This book also contains a glossary, a bibliography, and an index.

Highlighting and Operation Naming Conventions

The following highlighting conventions are used in this book with the noted exceptions:

Bold	Identifies menu choices, pushbuttons, commands, and shell script paths (except in reference information), default values, user selections, daemon paths (on first occurrence), and flags (in parameter lists).
Italics	Identifies parameters whose actual names or values are to be supplied by the user, and terms that are defined in the following text.
`Monospace`	Identifies subjects of examples, messages in text, examples of portions of program code, examples of text you might see displayed, information you should actually type, and examples used as teaching aids.

Terms and Abbreviations

This book uses the following terms:

DPI	Distributed Protocol Interface
MI	Management Interface
MIF	Management Information Format
RFC	An abbreviation for Internet Request for Comments documents
SNMP	Simple Network Management Protocol

Where to Find More Information

The SystemView Agent worldwide web page can be accessed through the following uniform resource locator (URL): http://www.software.ibm.com/ sysman/technology/caprod.html. See the web page for current updates and news about the product.

The following publications are included with the SystemView Agent program:

- *SystemView Agent User's Guide* , SVAG-USR2
- *SystemView Agent DMI Programmer's Guide* , SVAG-DMIP
- *SystemView Agent DPI Programmer's Guide* , SVAG-DPIP

The documents included with SystemView Agent are available in softcopy format. For OS/2 installations, use the VIEW facility.

SystemView Agent supports the most current DMI specification. At this time, this is DMI Version 1.1. The *Desktop Management Interface Specification, Version 1.1* is available from the Desktop Management Task Force.

Publications relevant to the SystemView Agent SNMP function are:

- RFC1592, which is the SNMP DPI 2.0 RFC.
- RFC1157, which describes SNMP Version 1
- RFC1901 through RFC1910, which describe SNMP Version 2

Other sources of information that can be helpful when using SystemView Agent are listed in Bibliography.

You can request IBM publications from your IBM representative or the IBM branch office serving your region. You can also contact the place where you purchased the SystemView Agent program.

Introduction to SNMP Distributed Protocol Interface

This section describes the SNMP DPI routines supported by SystemView Agent. This Application Programming Interface (API) is for the DPI

subagent programmer.

The reader may also want to obtain a copy of the relevant RFCs.

- RFC1592 is the SNMP DPI 2.0 RFC.
- RFC1901 through RFC1910 are the SNMP Version 2 RFCs.

Topics

SNMP Agents and Subagents

SNMP agents are responsible for answering SNMP requests from network management stations. Examples of management requests are GET, GETNEXT, and SET, performed on the MIB objects.

A subagent extends the set of MIB objects provided by the SNMP agent. With the subagent, you define MIB variables useful in your own environment and register them with the SNMP agent.

When the agent receives a request for a MIB variable, it passes the request to the subagent. The subagent then returns a response to the agent. The agent creates an SNMP response packet and sends the response to the remote network management station that initiated the request. The existence of the subagent is transparent to the network management station.

To allow the subagents to perform these functions, the agent provides for two types of subagent connections:

- TCP connection
- Connection via Shared Memory (SHM)

For the TCP connections, the agent binds to an arbitrarily chosen TCP port and listens for connection requests. A well-known port is not used. Every invocation of the SNMP agent could potentially use a different TCP port.

A subagent of the SNMP agent determines the port number by sending a GET request for an MIB variable, which represents the value of the TCP port. The subagent is not required to create and parse SNMP packets because the DPI API has a library routine query_DPI_port(). After the subagent obtains the value of the DPI TCP port, it should make a TCP connection to the appropriate port. After a successful connect(), the subagent registers the set of variables it supports with the SNMP agent. When all variable classes are registered, the subagent waits for requests from the SNMP agent.

The query_DPI_port() function is implicitly executed by the DPIconnect_to_agent_TCP() function. The DPI subagent programmer would normally use the DPIconnect_to_agent_TCP() function to connect to the agent, so it does not need to obtain the value of the DPI TCP port.

For a SHM connection, the subagent can use the DPIconnect_to_agent_SHM() function.

DPI Agent Requests

The SNMP agent can initiate several DPI requests:

- GET
- GETNEXT
- GETBULK (SNMP Version 2)
- SET, COMMIT, and UNDO
- UNREGISTER
- CLOSE

The GET, GETNEXT, GETBULK, and SET requests correspond to the SNMP requests that a network management station can make. The

subagent responds to a request with a response packet. The response packet can be created using the mkDPIresponse() library routine, which is part of the DPI API library.

The GETBULK requests are translated into multiple GETNEXT requests by the agent. According to RFC 1592, a subagent may request that the GETBULK be passed to it, but the OS/2 implementation of SystemView Agent does not support that request.

The COMMIT, UNDO, UNREGISTER, and CLOSE are specific SNMP DPI requests.

The subagent normally responds to a request with a RESPONSE packet. For the CLOSE and UNREGISTER request, the subagent does not need to send a RESPONSE.

Related Information

Overview of Subagent Processing
Connecting to the Agent
Registering a Sub-Tree with the Agent
Processing Requests from the Agent
Processing a GET Request
Processing a GETNEXT Request
Processing a SET/COMMIT/UNDO Request
Processing an UNREGISTER Request
Processing a CLOSE Request
Generating a TRAP

Multiple Levels of the SNMP DPI API

For the SNMP DPI 2.0 API, some functions are implemented as macros, because the older DPI Version 1.x had the same function names with different parameters. The new implementation has new function names, which are not always the most intuitive. By defining the macros with the more natural names for the functions, the non-intuitive names are hidden. This was done because the macros have the same names as the functions were named in DPI Version 1. It is thus possible to provide either the DPI 1.x or the DPI 2.x API by properly defining the macros.

SNMP DPI API Version 2.0

By default, when you include the snmp_dpi.h include file, you will be exposed to the DPI 2.0 API. For a list of the functions provided, see The snmp_dpi.h Include File. This is the recommended use of the SNMP DPI API. When you link-edit your object code into an executable file, you must use the new DPI functions as provided in the DPI20DLL.LIB and DPI20DLL.DLL files.

Waiting for a DPI packet also depends on the platform and how the chosen transport protocol is exactly implemented. In addition, some subagents want to control sending of and waiting for packets themselves, because they may need to be driven by other interrupts as well.

There is a set of DPI transport-related functions that are implemented on all platforms to hide the platform dependent issues for those subagents that do not need detailed control about the transport themselves.

Compatibility with DPI 1.x Base Code

If you have DPI 1.x based code, you may choose to keep using the old DPI 1.x dpi\snmp_dpi.h include file and the old DPI 1.x functions as provided in DPI32DLL.LIB and DPI32DLL.DLL. You will be able to communicate with the OS/2 SNMP agent which implements DPI 2.0.

In Migrating Your DPI Subagent to DPI 2.0, the changes that you must make to your DPI 1.x source are presented. If you take a few minutes to look at it, you will see that this is not too big a task.

SNMP DPI API Source Files

The following source files are provided:

snmp_dpi.h	The public SNMP DPI 2.0 API as provided to the DPI subagent programmer. The DPI subagent code must include this file.
dpi_samp.c	A very basic example of an SNMP DPI 2.0 subagent implementation, dpiSimp.mib.
dpiSimp.mib	The dpiSimple MIB that goes with the dpi_samp.c source.

The DPI subagent programmer can use the snmp_dpi.h include file and the dpi_samp.c file as an example of using the DPI API.

DPI Library

To use the DPI library routines provided with SystemView Agent, you must have the <snmp_dpi.h> header file in your SVA\DPI directory.

The DPI20DLL.LIB file in the SVA\DPI directory contains the DPI library routines. You must also have the SO32DLL.LIB and TCP32DLL.LIB files, which are provided by the OS/2 TCP/IP programmers toolkit.

You must define the OS2 variable to the compiler by doing one of the following:

- Place *#define OS2* at the top of each file that includes TCP/IP header files.
- Use the */DOS2* option when compiling the source for your application.

See the SVA\DPI directory for SNMP DPI sample programs.

Compiling, Linking, and Running a DPI API Application

Note: :The proper environment variables were set in the CONFIG.SYS file during installation.

The compiling and linking procedure for the DPI API using an IBM 32-bit compiler for OS/2 follows:

1. To compile the program, enter:

   ```
   icc /Sa /j- /Gm /C myprog.c
   ```

2. To create an executable program, you can enter:

   For VisualAge C++

   ```
   ilink /NOFREEFORMAT /De myprog,myprog.exe,NULL,so32dll.lib tcp32dll.lib
   os2386 dpi20dll.lib
   ```

   For C Set++

   ```
   link386 /De myprog,myprog.exe,NULL,so32dll.lib tcp32dll.lib
   os2386 dpi20dll.lib
   ```

For more information about the compile and link options, see the User's Guide provided with your compiler.

Functions, Data Structures, and Constants

Use these lists to locate the descriptions for the functions, data structures, and constants.

Basic DPI Functions:

DPI Transport-Related Functions:

Data Structures:

Constants and Values:

Related Information:

Subagent Programming Concepts

This section contains conceptual information about subagent programming.

Topics

Programming Recommendations

When implementing a subagent, it is recommended that you use the DPI Version 2 approach. This includes:

- Use the SNMP Version 2 error codes only, even though there are definitions for the SNMP Version 1 error codes.
- Implement the SET, COMMIT, UNDO processing properly.
- For GET requests, use the SNMP Version 2 approach and pass back noSuchInstance value or noSuchObject value if appropriate. Continue to process all remaining varBinds.
- For GETNEXT, use the SNMP Version 2 approach and pass back endOfMibView value if appropriate. Continue to process all remaining varBinds.
- When you are processing a request from the agent (GET, GETNEXT, GETBULK, SET, COMMIT, or UNDO), you are supposed to respond within the timeout period. You can specify the timeout period in the OPEN and REGISTER packets.

  If you fail to respond within the timeout period, the agent will probably close your DPI connection and then discard your RESPONSE packet if it comes in later. If you can detect that the response is not going to be received in the time period, then you might decide to stop the request and return an SNMP_ERROR_genErr in the RESPONSE.
- You may want to issue an SNMP DPI ARE_YOU_THERE request periodically to ensure that the agent is still "connected" and still knows about you.
- For OS/2, you use an ASCII based machine. However, when you are running on an EBCDIC based machine and you use the (default) native character set, then all OID strings and all variable values of type OBJECT_IDENTIFIER or DisplayString will be passed to you in EBCDIC format. OID strings include the group ID, instance ID, Enterprise ID, and subagent ID.

  When you return a response, you should then also use EBCDIC FORMAT.
- For OS/2, you use an ASCII based machine. However, when you are running on an EBCDIC based machine and you use the ASCII character set (specified in DPI OPEN), then all OID strings and all variable values of type OBJECT_IDENTIFIER or DisplayString will be passed to you in ASCII format. OID strings include the group ID, instance ID, Enterprise ID, and subagent ID.

  When you return a response, you should then also use ASCII FORMAT.
- If you receive an error RESPONSE on the OPEN packet, you will also receive a DPI CLOSE packet with an SNMP_CLOSE_openError code. In this situation, the agent closes the "connection".

  For OS/2, you use an ASCII based machine. However, when you connect to an EBCDIC based agent, you may want to specify in the DPI OPEN packet that you want to use ASCII character set on the agent. This is transparent to you and the burden of conversion is on the EBCDIC based agent.
- The DisplayString is only a textual convention. In the SNMP PDU (SNMP packet), the type is just an OCTET_STRING.

  When the type is OCTET_STRING, it is not clear if this is a DisplayString or any arbitrary data. This means that the agent can only know about an object being a DisplayString if the object is included in some sort of a compiled MIB. If it is, the agent will use SNMP_TYPE_DisplayString in the type field of the varBind in a DPI SET packet. When you send a DisplayString in a RESPONSE packet, the agent will handle it as such.

Related Information

A DPI Subagent Example

DPI API

The primary goal of RFC 1592 is to specify the SNMP DPI. This is a protocol by which subagents can exchange SNMP related information with an agent. On top of this protocol, one can imagine one or possibly many Application Programming Interfaces, but those are not addressed in RFC 1592.

In order to provide an environment that is generally platform independent, RFC 1592 strongly suggests that you also define a DPI API. There is a sample DPI API available in the RFC. The document describes the same sample API as the IBM supported DPI Version 2.0 API, see A DPI Subagent Example.

GET Processing

The DPI GET packet holds one or more varBinds that the subagent has taken responsibility for.

If the subagent encounters an error while processing the request, it creates a DPI RESPONSE packet with an appropriate error indication in the error_code field and sets the error_index to the position of the varBind at which the error occurs. The first varBind is index 1, the second varBind is index 2, and so on. No name, type, length, or value information needs to be provided in the packet because, by definition, the varBind information is the same as in the request to which this is a response and the agent still has that information.

If there are no errors, the subagent creates a DPI RESPONSE packet in which the error_code is set to SNMP_ERROR_noError (zero) and error_index is set to zero. The packet must also include the name, type, length, and value of each varBind requested.

When you get a request for a non-existing object or a non-existing instance of an object, you must return a NULL value with a type of SNMP_TYPE_noSuchObject or SNMP_TYPE_noSuchInstance respectively. These two values are not considered errors, so the error_code and error_index should be zero.

The DPI RESPONSE packet is then sent back to the agent.

Related Information

Processing a GET Request
The mkDPIresponse() Function

SET Processing

A DPI SET packet contains the name, type, length, and value of each varBind requested, plus the value type, value length, and value to be set.

If the subagent encounters an error while processing the request, it creates a DPI RESPONSE packet with an appropriate error indication in the error_code field and an error_index listing the position of the varBind at which the error occurs. The first varBind is index 1, the second varBind is index 2, and so on. No name, type, length, or value information needs to provided in the packet because, by definition, the varBind information is the same as in the request to which this is a response and the agent still has that information.

If there are no errors, the subagent creates a DPI RESPONSE packet in which the error_code is set to SNMP_ERROR_noError (zero) and error_index is set to zero. No name, type, length, or value information is needed because the RESPONSE to a SET should contain exactly the same varBind data as the data present in the request. The agent can use the values it already has.

This suggests that the agent must keep state information, and that is the case. It needs to do that anyway in order to be able to later pass the data with a DPI COMMIT or DPI UNDO packet. Since there are no errors, the subagent must have allocated the required resources and prepared itself for the SET. It does not yet carry out the set, that will be done at COMMIT time.

The subagent sends a DPI RESPONSE packet, indicating success or failure for the preparation phase, back to the agent. The agent will issue a SET request for all other varBinds in the same original SNMP request it received. This may be to the same subagent or to one or more different subagents.

Once all SET requests have returned a "no error" condition, the agent starts sending DPI COMMIT packets to the subagent(s). If any SET

request returns an error, the agent sends DPI UNDO packets to those subagents that indicated successful processing of the SET preparation phase.

When the subagent receives the DPI COMMIT packet, all the varBind information will again be available in the packet. The subagent can now carry out the SET request.

If the subagent encounters an error while processing the COMMIT request, it creates a DPI RESPONSE packet with value SNMP_ERROR_commitFailed in the error_code field and an error_index that lists at which varBind the error occurs. The first varBind is index 1, and so on. No name, type, length, or value information is needed. The fact that a commitFailed error exists does not mean that this error should be returned easily. A subagent should do all that is possible to make a COMMIT succeed.

If there are no errors and the SET and COMMIT have been carried out with success, the subagent creates a DPI RESPONSE packet in which the error_code is set to SNMP_ERROR_noError (zero) and error_index is set to zero. No name, type, length, or value information is needed.

So far we have discussed a successful SET and COMMIT sequence. However, after a successful SET, the subagent may receive a DPI UNDO packet. The subagent must now undo any preparations it made during the SET processing, such as free allocated memory.

Even after a COMMIT, a subagent may still receive a DPI UNDO packet. This will occur if some other subagent could not complete a COMMIT request. Because of the SNMP requirement that all varBinds in a single SNMP SET request must be changed "as if simultaneous", all committed changes must be undone if any of the COMMIT requests fail. In this case the subagent must try and undo the committed SET operation.

If the subagent encounters an error while processing the UNDO request, it creates a DPI RESPONSE packet with value SNMP_ERROR_undoFailed in the error_code field and an error_index that lists at which varBind the error occurs. The first varBind is index 1, and so on. No name, type, length, or value information is needed. The fact that an undoFailed error exists does not mean that this error should be returned easily. A subagent should do all that is possible to make an UNDO succeed.

If there are no errors and the UNDO has been successful, the subagent creates a DPI RESPONSE packet in which the error_code is set to SNMP_ERROR_noError (zero) and error_index is set to zero. No name, type, length, or value information is needed.

Related Information

Processing a SET/COMMIT/UNDO Request

GETNEXT Processing

The DPI GETNEXT packet contains the object(s) on which the GETNEXT operation must be performed. For this operation, the subagent is to return the name, type, length, and value of the next variable it supports whose (ASN.1) name lexicographically follows the one passed in the group ID (sub-tree) and instance ID.

In this case, the instance ID may not be present (NULL) in the incoming DPI packet implying that the NEXT object must be the first instance of the first object in the sub-tree that was registered.

It is important to realize that a given subagent may support several discontiguous sections of the MIB tree. In that situation, it would be incorrect to jump from one section to another. This problem is correctly handled by examining the group ID in the DPI packet. This group ID represents the "reason" why the subagent is being called. It holds the prefix of the tree that the subagent had indicated it supported (registered).

If the next variable supported by the subagent does not begin with that prefix, the subagent must return the same object instance as in the request, for example the group ID and instance ID with a value of SNMP_TYPE_endOfMibView (implied NULL value). This endOfMibView is not considered an error, so the error_code and error_index should be zero. If required, the SNMP agent will call upon the subagent again, but pass it a different group ID (prefix). This is illustrated in the discussion below.

Assume there are two subagents. The first subagent registers two distinct sections of the tree: A and C. In reality, the subagent supports variables A.1 and A.2, but it correctly registers the minimal prefix required to uniquely identify the variable class it supports.

The second subagent registers section B, which appears between the two sections registered by the first agent.

If a management station begins browsing the MIB, starting from A, the following sequence of queries of the form get-next(group ID,instance ID) would be performed:

```
Subagent 1 gets called:
       get-next(A,none) = A.1
       get-next(A,1)    = A.2
       get-next(A,2)    = endOfMibView
```

```
Subagent 2 is then called:
       get-next(B,none) = B.1
       get-next(B,1)    = endOfMibView

Subagent 1 gets called again:
       get-next(C,none) = C.1
```

Related Information

Processing a GETNEXT Request

GETBULK Processing

You can ask the agent to translate GETBULK requests into multiple GETNEXT requests. This is basically the default and is specified in the DPI REGISTER packet. In principle, we expect the majority of DPI subagents to run on the same machine as the agent, or on the same physical network. Therefore, repetitive GETNEXT requests remain local and, in general, should not be a problem.

Otherwise, the subagent can tell the agent to pass on a DPI GETBULK packet.

When a GETBULK request is received, the subagent must process the request and send a RESPONSE that sends back as many varBinds as requested by the request, as long as they fit within the buffers.

The GETBULK requires similar processing as a GETNEXT with regard to endOfMibView handling.

Note: A subagent cannot select GETBULK on OS/2. It will always be translated into multiple GETNEXT requests.

Related Information

Processing a GETNEXT Request

OPEN Request

As the first step, a DPI subagent must open a "connection" with the agent. To do so, it must send a DPI OPEN packet in which these parameters must be specified:

- The maximum timeout value in seconds. The agent is requested to wait this long for a response to any request for an object being handled by this subagent.

  The agent may have an absolute maximum timeout value which will be used if the subagent asks for too large a timeout value. A value of zero can be used to indicate that the agent's own default timeout value should be used. A subagent is advised to use a reasonably short interval of a few seconds or so. If a specific sub-tree needs a (much) longer time, a specific REGISTER can be done for that sub-tree with a longer timeout value.

- The maximum number of varBinds that the subagent is prepared to handle per DPI packet. Specifying 1 would result in DPI Version 1 behavior of one varBind per DPI packet that the agent sends to the subagent. A value of zero means the agent will try to combine up to as many varBinds as are present in the SNMP packet that belongs to the same sub-tree.

- The character set you want to use. By default, a 0 value, which is the native character set of the machine platform where the agent runs. Since the subagent and agent normally run on the same system or platform, you want to use the native character set, which is ASCII on many platforms.

  If your platform is EBCDIC based, using the native character set of EBCDIC makes it easy to recognize the string representations of the fields, such as the group ID and instance ID. At the same time, the agent will translate the value from ASCII NVT to EBCDIC and vice versa for objects that it knows from a compiled MIB to have a textual convention of DisplayString. This fact cannot be determined from the SNMP PDU encoding because in the PDU the object is only known to be an OCTET_STRING.

  If your subagent runs on an ASCII based platform and the agent runs on an EBCDIC based platform (or the other way around), you can specify that you want to use the ASCII character set. The agent and subagent programmer knows how to handle the string-based data in this situation.

Note: Not all agents need to support other than native character set selections. See Character Set Selection for more information on character set usage.

- The subagent ID. This an ASN.1 Object Identifier that uniquely identifies the subagent. This OID is represented as a null terminated string using the selected character set.

  For example: "1.3.5.1.2.3.4.5"

- The subagent description. This is a DisplayString describing the subagent. This is a character string using the selected character set.

  For example: "DPI sample subagent Version 2.0"

Once a subagent has sent a DPI OPEN packet to an agent, it should expect a DPI RESPONSE packet that informs the subagent about the result of the request. The packet ID of the RESPONSE packet should be the same as that of the OPEN request to which the RESPONSE packet is the response. See DPI RESPONSE Error Codes for a list of valid codes that may be expected.

If you receive an error RESPONSE on the OPEN packet, you will also receive a DPI CLOSE packet with an SNMP_CLOSE_openError code. In this situation, the agent closes the "connection".

If the OPEN is accepted, the next step is to REGISTER one or more MIB sub-trees.

Related Information

Connecting to the Agent

CLOSE Request

When a subagent is finished and wants to end processing, it should first UNREGISTER its sub-trees and then close the "connection" with the agent. To do so, it must send a DPI CLOSE packet, which specifies a reason for the closing. See DPI CLOSE Reason Codes for a list of valid codes. You should not expect a response to the CLOSE request.

A subagent should also be prepared to handle an incoming DPI CLOSE packet from the agent. In this case, the packet will contain a reason code for the CLOSE request. A subagent does not have to send a response to a CLOSE request. The agent just assumes that the subagent will handle it appropriately. The close takes place regardless of what the subagent does with it.

Related Information

Processing an CLOSE Request

REGISTER Request

Before a subagent will receive any requests for MIB variables, it must first register the variables or sub-tree it supports with the SNMP agent. The subagent must specify a number of parameters in the REGISTER request:

- The sub-tree to be registered. This is a null terminated string in the selected character set. The sub-tree must have a trailing dot.

  For example: "1.3.6.1.2.3.4.5."

- The requested priority for the registration. The values are:

-1	Request for the best available priority.
0	Request for the next best available priority than the highest (best) priority currently registered for this sub-tree.
NNN	Any other positive value requests that specific priority if available or the next worse priority that is available.

- The maximum timeout value in seconds. The agent is requested to wait this long for a response to any request for an object in this sub-tree. The agent may have an absolute maximum timeout value which will be used if the subagents asks for too large a timeout value. A value of zero can be used to indicate that the DPI OPEN value should be used for timeout.
- A specification if the subagent wants to do view selection. If it does, the community name from SNMP Version 1 packets will be passed in the DPI GET, GETNEXT, and SET packets. This is not supported on OS/2.
- A specification if the subagent wants to receive GETBULK packets or if it just prefers that the agent converts a GETBULK into multiple GETNEXT requests. This is not supported on OS/2.

Once a subagent has sent a DPI REGISTER packet to the agent, it should expect a DPI RESPONSE packet that informs the subagent about the result of the request. The packet ID of the RESPONSE packet should be the same as that of the REGISTER packet to which the RESPONSE packet is the response.

If the response is successful, the error_index field in the RESPONSE packet contains the priority that the agent assigned to the sub-tree registration. See DPI RESPONSE Error Codes for a list of valid codes that may be expected.

Error Code: higherPriorityRegistered

The response to a REGISTER request may return the error code "higherPriorityRegistered". This may be caused by:

- Another subagent already registered the same sub-tree at a better priority than what you are requesting.
- Another subagent already registered a sub-tree at a higher level (at any priority). For instance, if a registration already exists for sub-tree 1.2.3.4.5.6 and you try to register for sub-tree 1.2.3.4.5.6.<anything> then you will get "higherPriorityRegistered" error code.

If you receive this error code, your sub-tree will be registered, but you will not see any requests for the sub-tree. They will be passed to the sub-agent which registered with a better priority. If you stay connected, and the other sub-agent goes away, then you will get control over the sub-tree at that point in time.

Related Information

Registering a Sub-Tree with the Agent

UNREGISTER Request

A subagent may unregister a previously registered sub-tree. The subagent must specify a few parameters in the UNREGISTER request:

- The sub-tree to be unregistered. This is a null terminated string in the selected character set. The sub-tree must have a trailing dot.

  For example: "1.3.6.1.2.3.4.5."
- The reason for the unregister. See DPI UNREGISTER Reason Codes for a list of valid reason codes.

Once a subagent has sent a DPI UNREGISTER packet to the agent, it should expect a DPI RESPONSE packet that informs the subagent about the result of the request. The packet ID of the RESPONSE packet should be the same as that of the REGISTER packet to which the RESPONSE packet is the response. See DPI RESPONSE Error Codes for a list of valid codes that may be expected.

A subagent should also be prepared to handle incoming DPI UNREGISTER packets from the agent. In this situation, the DPI packet will contain a reason code for the UNREGISTER. A subagent does not have to send a response to an UNREGISTER request. The agent just assumes that the subagent will handle it appropriately. The registration is removed regardless of what the subagent returns.

Related Information

Processing an UNREGISTER Request

TRAP Request

A subagent can request that the SNMP agent generates a trap for it. The subagent must provide the desired values for the generic and

specific parameters of the trap. It may optionally provide a set of one or more name, type, length, or value parameters that will be included in the trap packet.

It may optionally specify an Enterprise ID (Object Identifier) for the trap to be generated. If a NULL value is specified for the Enterprise ID, the agent will use the subagent Identifier from the DPI OPEN packet as the Enterprise ID to be sent with the trap.

Related Information

Generating a TRAP

ARE_YOU_THERE Request

A subagent can send an ARE_YOU_THERE packet to the agent. This may be useful to do if you have a DPI "connection" over an unreliable transport protocol, such as UDP.

If the "connection" is in a healthy state, the agent responds with a RESPONSE packet with SNMP_ERROR_DPI_noError. If the "connection" is not in a healthy state, the agent may respond with a RESPONSE packet with an error indication, but the agent might not react at all. In this situation, you would timeout while waiting for a response.

Multithreading Programming Considerations

The DPI Version 2.0 DLL for OS/2 (DPI20DLL.DLL file) has been compiled with the /Gm+ compiler flag. This enables it to be used by both single and multithreaded subagents. However, even a single threaded subagent must be compiled with the /Gm+ compiler flag.

There are several static buffers in the DPI code. For compatibility reasons, that cannot be changed. Real multithread support will probably mean several potentially incompatible changes to the DPI 2.0 API.

Use a Locking Mechanism

If your subagent will be a multithreaded process, then you must always use some locking mechanism of your own around the use of the static buffers. Otherwise, one thread maybe writing into the static buffer while another is writing into the same buffer at the same time. There are two static buffers. One buffer is for building the serialized DPI packet before sending it out and the other buffer is for receiving incoming DPI packets.

Basically, all DPI functions that return a pointer to an unsigned char are the DPI functions that write into the static buffer to create a serialized DPI packet:

```
mkDPIAreYouThere()
mkDPIopen()
mkDPIregister()
mkDPIunregister()
mkDPItrap()
mkDPIresponse()
mkDPIpacket()
mkDPIclose ()
```

After you have called the DPIsend_packet_to_agent() function for the buffer, which is pointed to by the pointer returned by one of the above functions, it is free to use again.

There is one function that reads the static input buffer:

```
pDPIpacket()
```

The input buffer gets filled by the DPIawait_packet_from_agent() function. You get a pointer to the static input buffer upon return from the await. The pDPIpacket() function parses the static input buffer and returns a pointer to dynamically allocated memory. Therefore, after the pDPIparse() call, the buffer is available for use again.

The current situation is such that if multiple threads are waiting at the same time and for different handles, there is the risk that two incoming DPI packets will overlay each other.

If multiple threads are waiting for the same handle, when data arrives both threads come out of the wait. If one of them issues another wait before the other one is finished parsing the input buffer, the buffer may get overlaid by a new packet before the second one gets a chance to parse the packet.

The DPI internal handle structures and control blocks used by the underlying code to send and receive data to and from the agent are also static data areas. You must make sure that you use your own locking mechanism around the functions that add, change, or delete data in those static structures. The functions that change those internal static structures are:

```
DPIconnect_to_agentTCP()
DPIconnect_to_agentSHM()
DPIconnect_to_agentUDP()
DPIdisconnect_from_agent()
```

The other functions that access those static structures which must be assured that the structure is not being changed while they are referencing it during their execution are:

```
DPIawait_packet_from_agent()
DPIsend_packet_to_agent()
DPIget_fd_for_handle()
```

While the last 3 functions can be executed concurrently in different threads, you must ensure that no other thread is adding or deleting handles during this process.

Basic DPI API Functions

This section describes each of the basic DPI functions that are available to the DPI subagent programmer.

Topics

The DPIdebug() Function

Syntax

```
#include <snmp_dpi.h>

void DPIdebug(int level);
```

Parameters

level
: If this value is zero, tracing is turned off. If it has any other value, tracing is turned on at the specified level. The higher the value, the more detail. A higher level includes all lower levels of tracing. Currently there are two levels of detail:

1 Display packet creation and parsing.

2 Display hex dump of incoming and outgoing DPI packets.

Description

The DPIdebug() function turns DPI internal debugging/tracing on or off.

Examples

```
#include <snmp_dpi.h>

DPIdebug(2);
```

Related Information

The snmp_dpi.h Include File

The DPI_PACKET_LEN() Macro

Syntax

```
#include <snmp_dpi.h>

int DPI_PACKET_LEN(unsigned char *packet_p)
```

Parameters

packet_p
A pointer to a serialized DPI packet.

Return Values

An integer representing the total DPI packet length.

Description

The DPI_PACKET_LEN macro generates C-code that returns an integer representing the length of a DPI packet. It uses the first two octets in network byte order of the packet to calculate the length.

Examples

```
#include <snmp_dpi.h>
unsigned char *pack_p;
int           length;

pack_p = mkDPIclose(SNMP_CLOSE_goingDown);
if (pack_p) {
   length = DPI_PACKET_LEN(pack_p);
   /* send packet to agent or subagent */
} /* endif */
```

The fDPIparse() Function

Syntax

```
#include <snmp_dpi.h>

void fDPIparse(snmp_dpi_hdr *hdr_p);
```

Parameters

hdr_p
A pointer to the parse tree. The parse tree is represented by an snmp_dpi_hdr structure.

Description

The fDPIparse() function frees a parse tree that was previously created by a call to pDPIpacket(). The parse tree may have been created in other ways too. After calling fDPIparse(), no further references to the parse tree can be made.

A complete or partial DPI parse tree is also implicitly freed by call to a DPI function that serializes a parse tree into a DPI packet. The section that describes each function tells you if this is the case. An example of such a function is mkDPIresponse().

Examples

```
  #include <snmp_dpi.h>
  snmp_dpi_hdr  *hdr_p;
  unsigned char *pack_p;         /* assume pack_p points to  */
                                 /* incoming DPI packet      */
  hdr_p = pDPIpacket(pack_p);

  /* handle the packet and when done do the following */
  if (hdr_p) fDPIparse(hdr_p);
```

Related Information

The snmp_dpi_hdr Structure
The pDPIpacket() Function
The snmp_dpi.h Include File

The fDPIset() Function

Syntax

```
#include <snmp_dpi.h>

void fDPIset(snmp_dpi_set_packet *packet_p);
```

Parameters

packet_p
A pointer to the first snmp_dpi_set_packet structure in a chain of such structures.

Description

The fDPIset() function is typically used if you must free a chain of one or more snmp_dpi_set_packet structures. This may be the case if you are in the middle of preparing a chain of such structures for a DPI RESPONSE packet, but then run into an error before you can actually make the response.

If you get to the point where you make a DPI response packet to which you pass the chain of snmp_dpi_set_packet structures, then the mkDPIresponse() function will free the chain of snmp_dpi_set_packet structures.

Examples

```
#include <snmp_dpi.h>
unsigned char       *pack_p;
snmp_dpi_hdr        *hdr_p;
snmp_dpi_set_packet *set_p, *first_p;
long int             num1 = 0, num2 = 0;
```

```
hdr_p = pDPIpacket(pack_p);              /* assume pack_p     */
/* analyze packet and assume all OK */   /* points to the     */
/* now prepare response; 2 varBinds */   /* incoming packet   */

set_p = mkDPIset(snmp_dpi_NULL_p,        /* create first one  */
            "1.3.6.1.2.3.4.5.","1.0",    /* OID=1, instance=0 */
            SNMP_TYPE_Integer32,
            sizeof(num1), &num1);
if (set_p) {                             /* if success, then  */
   first_p = set_p;                      /* save ptr to first */
   set_p   = mkDPIset(set_p,             /* chain next one    */
               "1.3.6.1.2.3.4.5.","1.1", /* OID=1, instance=1 */
               SNMP_TYPE_Integer32,
               sizeof(num2), &num2);
   if (set_p) {                          /* success 2nd one   */
      pack_p = mkDPIresponse(hdr_p,      /* make response     */
                    SNMP_ERROR_noError,  /* It will also free */
                    0L, first_p);        /* the set_p tree    */
      /* send DPI response to agent */
   } else {                              /* 2nd mkDPIset fail */
      fDPIset(first_p);                  /* must free chain   */
   } /* endif */
} /* endif */
```

Related Information

The fDPIparse() Function
The snmp_dpi_set_packet Structure
The mkDPIresponse() Function

The mkDPIAreYouThere() Function

Syntax

```
#include <snmp_dpi.h>

unsigned char *mkDPIAreYouThere(void);
```

Parameters

None.

Return Values

If successful, a pointer to a static DPI packet buffer is returned. The first two bytes of the buffer in network byte order contain the length of the remaining packet. The macro DPI_PACKET_LEN can be used to calculate the total length of the DPI packet.

If failure, a NULL pointer is returned.

Note: The static buffer for the DPI packet is shared by other mkDPIxxxx() functions that create a serialized DPI packet.

Description

The mkDPIAreYouThere() function creates a serialized DPI ARE_YOU_THERE packet that can be sent to the DPI peer, which is normally the agent.

A subagent connected via TCP probably does not need this function because, normally when the agent breaks the "connection", you will receive an EOF on the file descriptor. For unreliable "connections", like over UDP, this function may be useful to periodically poll the agent and verify that it still knows about the subagent.

If your "connection" to the agent is still healthy, the agent will send a DPI RESPONSE with SNMP_ERROR_DPI_noError in the error code field and zero in the error index field. The RESPONSE will have no varBind data. If your "connection" is not healthy, the agent may send a response with an error indication, or may just not send a response at all.

Examples

```
#include <snmp_dpi.h>
unsigned char *pack_p;

pack_p = mkDPIAreYouThere();
if (pack_p) {
   /* send the packet to the agent */
} /* endif */
/* wait for response with DPIawait_packet_from_agent() */
/* normally the response should come back pretty quickly, */
/* but it depends on the load of the agent */
```

Related Information

The snmp_dpi_resp_packet Structure
The DPIawait_packet_from_agent() Function

The mkDPIclose() Function

Syntax

```
#include <snmp_dpi.h>

unsigned char *mkDPIclose(char reason_code);
```

Parameters

reason_code
The reason for closing the DPI connection. See DPI CLOSE Reason Codes for a list of valid reason codes.

Return Values

If successful, a pointer to a static DPI packet buffer is returned. The first two bytes of the buffer in network byte order contain the length of the remaining packet. The macro DPI_PACKET_LEN can be used to calculate the total length of the DPI packet.

If failure, a NULL pointer is returned.

Note: The static buffer for the DPI packet is shared by other mkDPIxxxx() functions that create a serialized DPI packet.

Description

The mkDPIclose() function creates a serialized DPI CLOSE packet that can be sent to the DPI peer. As a result of sending the packet, the DPI connection will be closed.

Sending a DPI CLOSE packet to the agent implies an automatic DPI UNREGISTER for all registered sub-trees on the connection being closed.

Examples

```
#include <snmp_dpi.h>
unsigned char *pack_p;

pack_p = mkDPIclose(SNMP_CLOSE_goingDown);
if (pack_p) {
   /* send the packet to the agent */
} /* endif */
```

Related Information

The snmp_dpi_close_packet Structure
DPI CLOSE Reason Codes

The mkDPIopen() Function

Syntax

```
#include <snmp_dpi.h>

unsigned char *mkDPIopen(       /* Make a DPI open packet     */
  char          *oid_p,         /* subagent Identifier (OID) */
  char          *description_p, /* subagent descriptive name */
  unsigned long  timeout,       /* requested default timeout  */
  unsigned long  max_varBinds,  /* max varBinds per DPI ndle  */
  char           character_set, /* selected character set     */
  #define DPI_NATIVE_CSET  0    /*   0 = native character set */
  #define DPI_ASCII_CSET   1    /*   1 = ASCII  character set */

  unsigned long  password_len,  /* length of password (if any)*/
  unsigned char *password_p);   /* ptr to password (if any)   */
```

Parameters

oid_p
: A pointer to a NULL terminated character string representing the OBJECT IDENTIFIER which uniquely identifies the subagent.

description_p
: A pointer to a NULL terminated character string, which is a descriptive name for the subagent. This can be any DisplayString, which basically is an octet string containing only characters from the ASCII NVT set.

timeout
: The requested timeout for this subagent. An agent often has a limit for this value and it will use that limit if this value is larger. A timeout of zero has a special meaning in the sense that the agent will use its own default timeout value.

max_varBinds
: The maximum number of varBinds per DPI packet that the subagent is prepared to handle. It must be a positive number or zero. If a value greater than 1 is specified, the agent will try to combine as many varBinds which belong to the same sub-tree per DPI packet as possible up to this value.

  If a value of zero is specified, the agent will try to combine up to as many varBinds as are present in the SNMP packet and belong to the same sub-tree. For example, a value of zero means no limit.

character_set
: The character set that you want to use for string-based data fields in the DPI packets and structures. The choices are:

DPI_NATIVE_CSET	Specifies that you want to use the native character set of the platform on which the agent that you connect to is running.
DPI_ASCII_CSET	Specifies that you want to use the ASCII character set. The agent will translate between ASCII and the native character set as required.

  See Character Set Selection for more information.

password_len
: The length in octets of an optional password. It depends on the agent implementation if a password is needed. If not, a zero length may be specified.

password_p
: A pointer to an octet string representing the password for this subagent. A password may include any character value, including the NULL character. If the password_len is zero, this can be a NULL pointer.

Return Values

If successful, a pointer to a static DPI packet buffer is returned. The first two bytes of the buffer in network byte order contain the length of the remaining packet. The macro DPI_PACKET_LEN can be used to calculate the total length of the DPI packet.

If failure, a NULL pointer is returned.

Note: The static buffer for the DPI packet is shared by other mkDPIxxxx() functions that create a serialized DPI packet.

Description

The mkDPIopen() function creates a serialized DPI OPEN packet that can then be sent to the DPI peer which is a DPI capable SNMP agent.

Normally you will want to use the native character set, which is the easiest for the subagent programmer. However, if the agent and subagent each run on their own platform and those platforms use different native character sets, you must select the ASCII character set, so that you both know exactly how to represent string-based data that is being send back and forth.

Currently you do not need to specify a password to connect to the OS/2 SNMP agent. Therefore, you can pass a length of zero and a NULL pointer for the password.

Examples

```
  #include <snmp_dpi.h>
  unsigned char *pack_p;

  pack_p = mkDPIopen("1.3.6.1.2.3.4.5",
                "Sample DPI subagent"
                0L,2L, DPI_NATIVE_CSET, /* max 2 varBinds */
                0,(char *)0);
  if (pack_p) {
     /* send packet to the agent */
  } /* endif */
```

Related Information

Character Set Selection

The mkDPIregister() Function

Syntax

```
#include <snmp_dpi.h>

unsigned char  *mkDPIregister(  /* Make a DPI register packet */
  unsigned short   timeout,     /* in seconds (16-bit)        */
  long int         priority,    /* requested priority         */
  char            *group_p,     /* ptr to group ID (sub-tree) */
  char             bulk_select);/* Bulk selection (GETBULK)   */
  #define DPI_BULK_NO   0       /*  map GETBULK into GETNEXTs */
  #define DPI_BULK_YES  1       /*  pass GETBULK to subagent */
```

Parameters

timeout
: The requested timeout in seconds. An agent often has a limit for this value and it will use that limit if this value is larger. The value zero has special meaning in the sense that it tells the agent to use the timeout value that was specified in the DPI OPEN packet.

priority
: The requested priority. This field may contain any of these values:

-1	Requests the best available priority.
0	Requests a better priority than the highest priority currently registered. Use this value to obtain the SNMP DPI Version 1 behavior.
nnn	Any positive value. You will receive that priority if available, otherwise the next best priority that is available.

group_p
: A pointer to a NULL terminated character string that represents the sub-tree to be registered. This group ID must have a trailing dot.

bulk_select

Specifies if you want the agent to pass GETBULK on to the subagent or to map them into multiple GETNEXT requests. The choices are:

DPI_BULK_NO	Do not pass any GETBULK requests, but instead map a GETBULK request into multiple GETNEXT requests.
DPI_BULK_YES	Do pass a GETBULK request to the subagent.

Return Values

If successful, a pointer to a static DPI packet buffer is returned. The first two bytes of the buffer in network byte order contain the length of the remaining packet. The macro DPI_PACKET_LEN can be used to calculate the total length of the DPI packet.

If failure, a NULL pointer is returned.

Note: The static buffer for the DPI packet is shared by other mkDPIxxxx() functions that create a serialized DPI packet.

Description

The mkDPIregister() function creates a serialized DPI REGISTER packet that can then be sent to the DPI peer which is a DPI capable SNMP agent.

The *bulk_select* can be used to ask the agent to map an SNMP GETBULK request into multiple GETNEXT requests. This makes it easier for the DPI subagent programmer because GETBULK processing doesn't need implementing.

However, if one expects that a single GETBULK might improve the performance a lot, one can tell the agent to pass such requests. This might be the case if one expects a GETBULK to arrive often for a table for which one needs to do a kernel dive. Using GETBULK, one might be able to do just one dive instead of many. Although one could anticipate the dive with a GETNEXT also, and therefore obtain and cache the table upon the first GETNEXT request.

According to the DPI 2.0 RFC, not all agents need to support DPI_BULK_YES. These agents will return an appropriate error code in the DPI RESPONSE though if such is the case.

Normally the SNMP agent sends a DPI RESPONSE packet back. This packet identifies if the register was successful or not.

Examples

```
#include <snmp_dpi.h>
unsigned char *pack_p;

pack_p = mkDPIregister(0,0L,"1.3.6.1.2.3.4.5."
                       DPI_BULK_NO);
if (pack_p) {
   /* send packet to agent and await response */
} /* endif */
```

Related Information

The snmp_dpi_resp_packet Structure

The mkDPIresponse() Function

Syntax

```
#include <snmp_dpi.h>

unsigned char    *mkDPIresponse( /* Make a DPI response packet*/
  snmp_dpi_hdr     *hdr_p,       /* ptr to packet to respnd to*/
  long int          error_code,  /* error code: SNMP_ERROR_xxx*/
  long int          error_index, /* index to varBind in error */
  snmp_dpi_set_packet *packet_p);/* ptr to varBinds, a chain  */
                                 /* of dpi_set_packets        */
```

Parameters

hdr_p
A pointer to the parse tree of the DPI request to which this DPI packet will be the response. The function uses this parse tree to copy the packet_id and the DPI version and release, so that the DPI packet is correctly formatted as a response.

error_code
The error code.

See DPI RESPONSE Error Codes for a list of valid codes.

error_index
Specifies the first varBind in error. Counting starts at 1 for the first varBind. This field should be zero if there is no error.

packet_p
A pointer to a chain of snmp_dpi_set_packet structures. This partial parse tree will be freed by the mkDPIresponse() function. So upon return you cannot reference it anymore. Pass a NULL pointer if there are no varBinds to be returned.

Return Values

If successful, a pointer to a static DPI packet buffer is returned. The first two bytes of the buffer in network byte order contain the length of the remaining packet. The macro DPI_PACKET_LEN can be used to calculate the total length of the DPI packet.

If failure, a NULL pointer is returned.

Note: The static buffer for the DPI packet is shared by other mkDPIxxxx() functions that create a serialized DPI packet.

Description

The mkDPIresponse() function is used at the subagent side to prepare a DPI RESPONSE packet to a GET, GETNEXT, GETBULK, SET, COMMIT or UNDO request. The resulting packet can be sent to the DPI peer, which is normally a DPI capable SNMP agent.

Examples

```
#include <snmp_dpi.h>
unsigned char       *pack_p;
snmp_dpi_hdr        *hdr_p;
snmp_dpi_set_packet *set_p;
long int             num;

hdr_p = pDPIpacket(pack_p);     /* parse incoming packet */
                                /* assume it's in pack_p */
if (hdr_p) {
   /* analyze packet, assume GET, no error */
   set_p = mkDPIset(snmp_dpi_set_packet_NULL_p,
                    "1.3.6.1.2.3.4.5.", "1.0",
                    SNMP_TYPE_Integer32,
                    sizeof(num), &num);
   if (set_p) {
      pack_p = mkDPIresponse(hdr_p,
                    SNMP_ERROR_noError, 0L, set_p);
      if (pack_p) {
         /* send packet to subagent */
      } /* endif */
   } /* endif */
} /* endif */
```

The mkDPIresponse() function is used at the agent side to prepare a DPI RESPONSE packet to an OPEN, REGISTER or UNREGISTER request. In the case of a RESPONSE to a REGISTER request and if there is no error, the actually assigned priority must be passed in the *error_index* parameter. The resulting packet can be sent to the DPI peer, which is normally a subagent.

Examples

```
#include <snmp_dpi.h>
unsigned char       *pack_p;
snmp_dpi_hdr        *hdr_p;
long int             priority;

hdr_p = pDPIpacket(pack_p);     /* parse incoming packet */
                                /* assume it's in pack_p */
if (hdr_p) {
   /* analyze packet, assume REGISTER and OK */
   pack_p = mkDPIresponse(hdr_p,
                SNMP_ERROR_DPI_noError,
                priority,
                snmp_dpi_set_packet_NULL_p);
```

```
    if (pack_p) {
       /* send packet to subagent */
    } /* endif */
  } /* endif */
```

Related Information

The pDPIpacket() Function
The snmp_dpi_hdr Structure
The snmp_dpi_next_packet Structure

The mkDPIset() Function

Syntax

```
#include <snmp_dpi.h>

snmp_dpi_set_packet *mkDPIset(    /* Make DPI set packet tree */
  snmp_dpi_set_packet *packet_p,  /* ptr to SET structure     */
  char                *group_p,   /* ptr to group ID(sub-tree)*/
  char                *instance_p,/* ptr to instance OIDstring*/
  int                  value_type,/* value type: SNMP_TYPE_xxx*/
  int                  value_len, /* length of value          */
  void                *value_p);  /* ptr to value             */
```

Parameters

packet_p
A pointer to a chain of snmp_dpi_set_packet structures. Pass a NULL pointer if this is the first structure to be created.

group_p
A pointer to a NULL terminated character string that represents the registered sub-tree that caused this GET request to be passed to this DPI subagent. The sub-tree must have a trailing dot.

instance_p
A pointer to a NULL terminated character string that represents the rest, which is the piece following the sub-tree part, of the OBJECT IDENTIFIER of the variable instance being accessed. Use of the term *instance_p* here should not be confused with an OBJECT instance because this string may consist of a piece of the OBJECT IDENTIFIER plus the INSTANCE IDENTIFIER.

value_type
The type of the value.

See DPI SNMP Value Types for a list of currently defined value types.

value_len
This is the value that specifies the length in octets of the value pointed to by the *value* field. The length may be zero if the value is of type SNMP_TYPE_NULL.

The maximum value is 64K -1. However, the implementation often makes the length significantly less. The OS/2 implementation limit is 4K. The SNMP_DPI_BUFFSIZE in the snmp_dpi.h include file defines the limit for OS/2.

value_p
A pointer to the actual value. This field may contain a NULL pointer if the value is of implicit or explicit type SNMP_TYPE_NULL.

Return Values

If successful and a chain of one or more packets was passed in the *packet_p* parameter, the same pointer that was passed in *packet_p* is returned. A new dynamically allocated structure has then been added to the end of that chain of snmp_dpi_get_packet structures.

If successful and a NULL pointer was passed in the *packet_p* parameter, a pointer to a new dynamically allocated structure is returned.

If failure, a NULL pointer is returned.

Description

The mkDPIset() function is used at the subagent side to prepare a chain of one or more snmp_dpi_set_packet structures. This chain is used to create a DPI RESPONSE packet by a call to mkDPIresponse() which can be sent to the DPI peer, which is normally a DPI capable SNMP agent.

The chain of snmp_dpi_set_packet structures can also be used to create a DPI TRAP packet that includes varBinds as explained in The mkDPItrap() Function.

For the value_len, the maximum value is 64K -1. However, the implementation often makes the length significantly less. For example the SNMP PDU size may be limited to 484 bytes at the SNMP manager or agent side. In this case, the total response packet cannot exceed 484 bytes, so a value_len is limited by that. You can send the DPI packet to the agent, but the manager will never see it.

Examples

```
#include <snmp_dpi.h>
unsigned char       *pack_p;
snmp_dpi_hdr        *hdr_p;
snmp_dpi_set_packet *set_p;
long int             num;

hdr_p = pDPIpacket(pack_p)      /* parse incoming packet */
                                /* assume it's in pack_p */
if (hdr_p) {
   /* analyze packet, assume GET, no error */
   set_p = mkDPIset(snmp_dpi_set_packet_NULL_p,
                    "1.3.6.1.2.3.4.5.", "1.0",
                    SNMP_TYPE_Integer32,
                    sizeof(num), &num);
   if (set_p) {
      pack_p = mkDPIresponse(hdr_p,
                    SNMP_ERROR_noError,
                    0L, set_p);
      if (pack_p)
         /* send packet to subagent */
      } /* endif */
   } /* endif */
} /* endif */
```

The mkDPIset() function is used at the agent side to prepare a chain of one or more snmp_dpi_set_packet structures. This chain is normally anchored in an snmp_dpi_hdr structure that has its *packet_type* field set to SNMP_DPI_SET, SNMP_DPI_COMMIT or SNMP_DPI_UNDO. When all varBinds have been prepared into snmp_dpi_set_packet structures, a call can be made to mkDPIpacket() which will serialize the DPI parse tree into a DPI packet that can be sent to the DPI peer, which is normally a subagent.

Examples

```
#include <snmp_dpi.h>
unsigned char       *pack_p;
snmp_dpi_hdr        *hdr_p;
long int             num;

hdr_p = mkDPIhdr(SNMP_DPI_SET);
if (hdr_p) {
   hdr_p->data_u.set_p =
          mkDPIset(snmp_dpi_set_packet_NULL_p,
                   "1.3.6.1.2.3.4.5.", "1.0",
                   SNMP_TYPE_Integer32,
                   sizeof(num), &num);
   if (hdr_p->data_u.set_p) {
      pack_p = mkDPIpacket(hdr_p);
      if (pack_p)
         /* send packet to subagent */
      } /* endif */
   } /* endif */
} /* endif */
```

If you must chain many snmp_dpi_set_packet structures, be sure to note that the packets are chained only by forward pointers. It is recommended that you use the last structure in the existing chain as the *packet_p* parameter. Then, the underlying code does not have to scan through a possibly long chain of structures in order to chain the new structure at the end.

In the next example let's assume that we want to chain 20 snmp_dpi_set_packet structures as a response to a GETBULK.

Examples

```
  #include <snmp_dpi.h>
  unsigned char       *pack_p;
  snmp_dpi_hdr        *hdr_p;
  snmp_dpi_set_packet *first_p;
  snmp_dpi_set_packet *set_p;
  long int             num[20];
  int                  i;

  hdr_p = pDPIpacket(pack_p);    /* parse incoming packet    */
                                 /* assume it's in pack_p    */
  if (hdr_p) {
     /* analyze packet, assume GETBULK, no error. In this    */
     /* example we do not check max_repetitions as we should */
     set_p   = snmp_dpi_set_packet_NULL_p;
     first_p = snmp_dpi_set_packet_NULL_p;
     for (i=0; i<20; i++) {
         char instance[5];

         sprintf(instance, "%1.%d", i+1);
         set_p = mkDPIset(set_p,
                     "1.3.6.1.2.3.4.5.", instance,
                     SNMP_TYPE_Integer32,
                     sizeof(num), &num[i]);
         if (set_p) {
            if (first_p) continue; /* OK, iterate for loop  */
            first_p = set_p;       /* remember first one    */
         } else if (first_p) {     /* failed to mkDPIset    */
            fDPIset(first_p)       /* free allocated memory */
            first_p = snmp_dpi_set_packet_NULL_p; /* reset  */
         } /* endif */
     } /* endfor */
     if (first_p) {
        pack_p = mkDPIresponse(hdr_p,
                     SNMP_ERROR_noError,
                     0L, first_p);
        if (pack_p)
           /* send packet to subagent */
        } /* endif */
     } /* endif */
  } /* endif */
```

Related Information

The pDPIpacket() Function
The mkDPIresponse() Function
The mkDPItrap() Function
The snmp_dpi_hdr Structure
The snmp_dpi_set_packet Structure
DPI SNMP Value Types
Value Representation

The mkDPItrap() Function

Syntax

```
#include <snmp_dpi.h>

unsigned char      *mkDPItrap(   /* Make a DPI trap packet    */
  long int            generic,   /* generic traptype  (32 bit)*/
  long int            specific,  /* specific traptype (32 bit)*/
  snmp_dpi_set_packet *packet_p, /* ptr to varBinds, a chain  */
                                 /* of dpi_set_packets        */
  char           *enterprise_p); /* ptr to enterprise OID     */
```

Parameters

generic
The generic trap type. The range of this value is 0-6, where 6, which is enterprise specific, is the type that is probably used most by DPI subagent programmers. The values 0-5 are well defined standard SNMP traps.

specific
The enterprise specific trap type. This can be any value that is valid for the MIB sub-trees that the subagent implements.

packet_p
A pointer to a chain of snmp_dpi_set_structures, representing the varBinds to be passed with the trap. This partial parse tree will be freed by the mkDPItrap() function so you cannot reference it anymore upon completion of the call. A NULL pointer means that there are no varBinds to be included in the trap.

enterprise_p
A pointer to a NULL terminated character string representing the enterprise ID (OBJECT IDENTIFIER) for which this trap is defined. A NULL pointer can be used. In this case, the subagent Identifier, as passed in the DPI OPEN packet, will be used when the agent receives the DPI TRAP packet.

Return Values

If successful, a pointer to a static DPI packet buffer is returned. The first two bytes of the buffer in network byte order contain the length of the remaining packet. The macro DPI_PACKET_LEN can be used to calculate the total length of the DPI packet.

If failure, a NULL pointer is returned.

Note: The static buffer for the DPI packet is shared by other mkDPIxxxx() functions that create a serialized DPI packet.

Description

The mkDPItrap() function is used at the subagent side to prepare a DPI TRAP packet. The resulting packet can be sent to the DPI peer, which is normally a DPI capable SNMP agent.

Examples

```
  #include <snmp_dpi.h>
  unsigned char       *pack_p;
  snmp_dpi_set_packet *set_p;
  long int             num;

  set_p = mkDPIset(snmp_dpi_set_packet_NULL_p,
                   "1.3.6.1.2.3.4.5.", "1.0",
                   SNMP_TYPE_Integer32,
                   sizeof(num), &num);
  if (set_p) {
     pack_p = mkDPItrap(6,1,set_p, (char *)0);
     if (pack_p) {
        /* send packet to subagent */
     } /* endif */
  } /* endif */
```

Related Information

The fDPIset() Function

The mkDPIunregister() Function

Syntax

```
#include <snmp_dpi.h>

unsigned char *mkDPIunregister( /* Make DPI unregister packet */
  char           reason_code;   /* unregister reason code     */
  char          *group_p);      /* ptr to group ID (sub-tree) */
```

Parameters

reason_code
The reason for the unregister.

See DPI UNREGISTER Reason Codes for a list of the currently defined reason codes.

group_p
A pointer to a NULL terminated character string that represents the sub-tree to be unregistered. The sub-tree must have a trailing dot.

Return Values

If successful, a pointer to a static DPI packet buffer is returned. The first two bytes of the buffer in network byte order contain the length of the remaining packet. The macro DPI_PACKET_LEN can be used to calculate the total length of the DPI packet.

If failure, a NULL pointer is returned.

Note: The static buffer for the DPI packet is shared by other mkDPIxxxx() functions that create a serialized DPI packet.

Description

The mkDPIunregister() function creates a serialized DPI UNREGISTER packet that can be sent to the DPI peer, which is a DPI capable SNMP agent or subagent.

Normally, the SNMP peer then sends a DPI RESPONSE packet back. This packet identifies if the unregister was successful or not.

Examples

```
#include <snmp_dpi.h>
unsigned char *pack_p;

pack_p = mkDPIunregister(
              SNMP_UNREGISTER_goingDown,
              "1.3.6.1.2.3.4.5.");
if (pack_p) {
   /* send packet to agent or subagent and await response */
} /* endif */
```

Related Information

The snmp_dpi_ureg_packet Structure

The pDPIpacket() Function

Syntax

```
#include <snmp_dpi.h>

snmp_dpi_hdr *pDPIpacket(unsigned char *packet_p);
```

Parameters

packet_p
A pointer to a serialized DPI packet.

Return Values

If successful, a pointer to a DPI parse tree (snmp_dpi_hdr) is returned. Memory for the parse tree has been dynamically allocated, and it is the callers responsibility to free it when no longer needed. You can use the fDPIparse() function to free the parse tree.

If failure, a NULL pointer is returned.

Description

The pDPIpacket() function parses the buffer pointed to by the *packet_p* parameter. It ensures that the buffer contains a valid DPI packet

and that the packet is for a DPI version and release that is supported by the DPI functions in use.

Examples

```
  #include <snmp_dpi.h>
  unsigned char       *pack_p;
  snmp_dpi_hdr        *hdr_p;

  hdr_p = pDPIpacket(pack_p);       /* parse incoming packet */
                                    /* assume it's in pack_p */
  if (hdr_p) {
     /* analyze packet, and handle it */
  }
```

Related Information

The snmp_dpi_hdr Structure
The snmp_dpi.h Include File
The fDPIparse() Function

Transport-Related DPI API Functions

This section describes each of the DPI transport-related functions that are available to the DPI subagent programmer. These functions try to hide any platform specific issues for the DPI subagent programmer so that the subagent can be made as portable as possible. If you need detailed control for sending and awaiting DPI packets, you may have to do some of the transport-related code yourself.

The transport-related functions are basically the same for any platform, except for the initial call to setup a connection. OS/2 currently supports the TCP/IP transport type.

Topics

The DPIawait_packet_from_agent() Function
The DPIconnect_to_agent_SHM() Function
The DPIconnect_to_agent_TCP() Function
The DPIdisconnect_from_agent() Function
The DPIget_fd_for_handle() Function
The DPIsend_packet_to_agent() Function
The lookup_host() Function
The query_DPI_port() Function

The DPIawait_packet_from_agent() Function

Syntax

```
#include <snmp_dpi.h>

int DPIawait_packet_from_agent(    /* await a DPI packet      */
  int                   handle,    /* on this connection      */
  int                   timeout,   /* timeout in seconds      */
  unsigned char       **message_p, /* receives ptr to data    */
  unsigned long        *length);   /* receives length of data */
```

Parameters

handle
A handle as obtained with a DPIconnect_to_agent_xxxx() call.

timeout

A timeout value in seconds. There are two special values:

-1	Causes the function to wait forever until a packet arrives.
0	Means that the function will only check if a packet is waiting. If not, an immediate return is made. If there is a packet, it will be returned.

message_p
The address of a pointer that will receive the address of a static DPI packet buffer or, if there is no packet, a NULL pointer.

length
The address of an unsigned long integer that will receive the length of the received DPI packet or, if there is no packet, a zero value.

Return Values

If successful, a zero (DPI_RC_noError) is returned. The buffer pointer and length of the caller will be set to point to the received DPI packet and to the length of that packet.

If failure, a negative integer is returned. It indicates the kind of error that occurred. See Return Codes from DPI Transport-Related Functions for a list of possible error codes.

Description

The DPIawait_packet_from_agent() function is used at the subagent side to await a DPI packet from the DPI capable SNMP agent. The programmer can specify how long to wait.

Examples

```
#include <snmp_dpi.h>
int                 handle;
unsigned char      *pack_p;
unsigned long       length;

handle = DPIconnect_to_agent_TCP("localhost", "public");
if (handle < 0) {
   printf("Error %d from connect\n",handle);
   exit(1);
} /* endif */
/* do useful stuff */
rc = DPIawait_packet_from_agent(handle, -1,
                          &pack_p, &length);
if (rc) {
   printf("Error %d from await packet\n");
   exit(1);
} /* endif */
/* handle the packet */
```

Related Information

The DPIconnect_to_agent_TCP() Function

The DPIconnect_to_agent_SHM() Function

Syntax

```
#inlcude <snmp_dpi.h>

int  DPIconnect_to_agent_SHM(  /* Connect to DPI Shared Mem  */
        int          queue_id);/* target (agent) queue id    */
```

Parameters

queu_id
A queue_id known by the agent. The value is a fixed queueid. It must always be 1.

Return Values

If successful, a positive integer that represents the connection is returned. It is to be used as a handle in subsequent calls to DPI transport-related functions.

If failure, a negative integer is returned. It indicates the kind of error that occurred. See Return Codes from DPI Transport-Related Functions for a list of possible error codes.

Description

The DPIconnect_to_agent_SHM() function is used at the subagent side to setup a connection (via SHared Memory) to the DPI capable SNMP agent.

Examples

```
#include <snmp_dpi.h>
int                 handle;

handle = DPIconnect_to_agent_SHM(1);
if (handle < 0) {
   printf("Error %d from connect\n",handle);
   exit(1);
}          /* endif */
```

Related Information

Return Codes from DPI Transport-Related Functions
The DPIconnect_to_agent_TCP() Function

The DPIconnect_to_agent_TCP() Function

Syntax

```
#include <snmp_dpi.h>

int DPIconnect_to_agent_TCP(    /* Connect to DPI TCP port    */
  char          *hostname_p,    /* target hostname/IP address */
  char          *community_p);  /* community name             */
```

Parameters

hostname_p
A pointer to a NULL terminated character string representing the host name or IP address in dot notation of the host where the DPI capable SNMP agent is running.

community_p
A pointer to a NULL terminated character string representing the community name that is required to obtain the dpiPort from the SNMP agent via an SNMP GET request.

Return Values

If successful, a positive integer that represents the connection is returned. It is to be used as a handle in subsequent calls to DPI transport-related functions.

If failure, a negative integer is returned. It indicates the kind of error that occurred. See Return Codes from DPI Transport-Related Functions for a list of possible error codes.

Description

The DPIconnect_to_agent_TCP() function is used at the subagent side to setup a TCP connection to the DPI capable SNMP agent.

Examples

```
  #include <snmp_dpi.h>
  int                 handle;
```

```
handle = DPIconnect_to_agent_TCP("localhost", "loopback");
if (handle < 0) {
   printf("Error %d from connect\n",handle);
   exit(1);
} /* endif */
```

Related Information

Return Codes from DPI Transport-Related Functions
The DPIconnect_to_agent_SHM() Function

The DPIdisconnect_from_agent() Function

Syntax

```
#include <snmp_dpi.h>

void DPIdisconnect_from_agent(  /* disconnect from DPI (agent)*/
  int                  handle); /* close this connection      */
```

Parameters

handle
A handle as obtained with a DPIconnect_to_agent_xxxx() call.

Return Values

If successful, a positive integer that represents the connection is returned. It is to be used as a handle in subsequent calls to DPI transport-related functions.

If failure, a negative integer is returned. It indicates the kind of error that occurred. See Return Codes from DPI Transport-Related Functions for a list of possible error codes.

Description

The DPIdisconnect_from_agent() function is used at the subagent side to terminate a connection to the DPI capable SNMP agent.

Examples

```
#include <snmp_dpi.h>
int                  handle;

handle = DPIconnect_to_agent_TCP("localhost", "loopback");
if (handle < 0) {
   printf("Error %d from connect\n",handle);
   exit(1);
} /* endif */
/* do useful stuff */
DPIdisconnect_from_agent(handle);
```

Related Information

The DPIconnect_to_agent_TCP() Function

The DPIget_fd_for_handle() Function

Syntax

```
#include <snmp_dpi.h>

int DPIget_fd_for_handle(       /* get the file descriptor    */
  int            handle);       /* for this handle            */
```

Parameters

handle
A handle that was obtained with a DPIconnect_to_agent_TCP() call.

Return Values

If successful, a positive integer representing the file descriptor associated with the specified handle.

If failure, a negative integer is returned. It indicates the error that occurred. See Return Codes from DPI Transport-Related Functions for a list of possible error codes.

Description

The DPIget_fd_for_handle function is used to obtain the file descriptor for the handle, which was obtained with a DPIconnect_to_agent_TCP() call.

The DPI subagent programmer would use this function to not only wait for DPI requests, but possibly for other TCP/IP events. The programmer may want to do their own select and include for the file descriptor of the DPI connections.

Examples

```
#include <snmp_dpi.h>
#include /* other include files for BSD sockets and such */
int                 handle;
int                 fd;

handle = DPIconnect_to_agentTCP("localhost","public");
if (handle < 0) {
   printf("Error %d from connect\n",handle);
   exit(1);
}
fd = DPIget_fd_for_handle(handle);
if (fd <0) {
   printf("Error %d from get_fd\n",fd);
   exit(1);
}
```

Related Information

The DPIconnect_to_agent_TCP() Function

The DPIsend_packet_to_agent() Function

Syntax

```
#include <snmp_dpi.h>

int DPIsend_packet_to_agent(       /* send  a DPI packet      */
  int                  handle,    /* on this connection      */
  unsigned char        *message_p, /* ptr to the packet data  */
  unsigned long         length);   /* length of the packet    */
```

Parameters

handle
A handle as obtained with a DPIconnect_to_agent_xxxx() call.

message_p
A pointer to the buffer containing the DPI packet to be sent.

length
The length of the DPI packet to be sent. The DPI_PACKET_LEN macro is a useful macro to calculate the length.

Return Values

If successful, a zero (DPI_RC_noError) is returned.

If failure, a negative integer is returned. It indicates the kind of error that occurred. See Return Codes from DPI Transport-Related Functions for a list of possible error codes.

Description

The DPIsend_packet_to_agent() function is used at the subagent side to send a DPI packet to the DPI capable SNMP agent.

Examples

```
  #include <snmp_dpi.h>
  int                 handle;
  unsigned char      *pack_p;

  handle = DPIconnect_to_agent_TCP("localhost", "public");
  if (handle < 0) {
     printf("Error %d from connect\n",handle);
     exit(1);
  } /* endif */
  pack_p = mkDPIopen("1.3.6.1.2.3.4.5",
                "Sample DPI subagent"
                0L,2L,,DPI_NATIVE_CSET,
                0,(char *)0);
  if (pack_p) {
     rc = DPIsend_packet_to_agent(handle,pack_p,
                        DPI_PACKET_LEN(pack_p));
     if (rc) {
        printf("Error %d from await packet\n");
        exit(1);
     } /* endif */
  } else {
     printf("Can't make DPI OPEN packet\n");
     exit(1);
  } /* endif */
  /* await the response */
```

Related Information

The DPIconnect_to_agent_TCP() Function
The DPI_PACKET_LEN() Macro
The mkDPIopen() Function

The lookup_host() Function

Syntax

```
#include <snmp_dpi.h>

unsigned long    lookup_host(   /* find IP address in network */
  char            *hostname_p); /* byte order for this host   */
```

Parameters

hostname_p
A pointer to a NULL terminated character string representing the host name or IP address in dot notation of the host where the DPI capable SNMP agent is running.

Return Values

If successful, the IP address is returned in network byte order, so it is ready to be used in a sockaddr_in structure.

If failure, a value of 0 is returned.

Description

The lookup_host() function is used to obtain the IP address in network byte order of a host or IP address in dot notation.

The DPI subagent programmer only needs to use this function to code the connection setup and send or await the packet. The programmer then obtains the DPI port number, finds the IP address of the agent with the lookup_host() function and then sets up a socket for communication.

This function is implicitly executed by the DPIconnect_to_agent_TCP() function, which is the function that the DPI subagent programmer would normally use. So the lookup_host() function is normally not used by the DPI subagent programmer.

Examples

```
#include <snmp_dpi.h>
#include /* other include files for BSD sockets and such */
int                 handle;
unsigned char      *pack_p;
long int            dpi_port;
int                 fd;
struct sockaddr_in  s,t;             /* source and target   */

dpi_port = query_DPI_port("localhost", /* get DPI port number */
                 "public",           /* for TCP, local host */
                 dpiPortForTCP);
if (dpi_port < 0) exit(1);           /* error if negative   */

host_addr = lookup_host("localhost");  /* find target IP addr */
if (host_addr == 0) exit(1);           /* unknown, that's it  */

fd = socket(AF_INET,SOCK_STREAM,0);    /* create a TCP socket */
if (fd < 0) exit(1);                   /* failure to do so    */

memset(&s,0,sizeof(s));
s.sin_family      = AF_INET;           /* set AF_INET family  */
s.sin_port        = 0;                 /* give us any port,   */
s.sin_addr.s_addr = htonl(INADDR_ANY); /* any local IPaddress */

rc = bind(fd,(struct sockaddr *)s,     /* bind our socket(fd) */
          sizeof(s_sock));             /* defined in s socket */
if (rc < 0) exit(1);                   /* failure, so exit    */

memset(&d,0,sizeof(d));
d.sin_family      = AF_INET;           /* set AF_INET family  */
d.sin_port        = htons(dpi_port);   /* set requested port  */
d.sin_addr.s_addr = host_addr;         /* destination IP addr */
                                       /* network byte order  */
rc = connect(fd,(struct sockaddr *)d,  /* connect to target   */
             sizeof(d));               /* based on d sock     */
if (rc < 0) exit(1);                   /* failed, exit        */
/* now we have a socket on which to send/receive DPI packets  */
```

Related Information

The query_DPI_port() Function
The DPIconnect_to_agent_TCP() Function

The query_DPI_port() Function

Syntax

```
#include <snmp_dpi.h>

long int query_DPI_port(        /* Query (GET) SNMP_DPI port  */
  char          *hostname_p,    /* target hostname/IPaddress  */
  char          *community_p,   /* communityname for GET      */
  int            porttype);     /* port type, one of:         */
                                /*   dpiPortForTCP            */
                                /*   dpiPortForUDP            */
```

Parameters

hostname_p
: A pointer to a NULL terminated character string representing the host name or IP address in dot notation of the host where the DPI capable SNMP agent is running.

community_p
: A pointer to a NULL terminated character string representing the community name that is required to obtain the dpiPort from the SNMP agent via an SNMP GET request.

porttype
: The dpiPort object for a specific port type that you want to obtain. Currently there are two types: one for a TCP port and one for a UDP port. The snmp_dpi.h include file has two #define statements for these DPI port types:

```
#define dpiPortForTCP    1
#define dpiPortForUDP    2
```

At this time, the dpiPORTForUDP port type is not supported. If you use it, the return value is set to -1, which indicates a failure.

Return Values

If successful, the DPI port number for the specified protocol, TCP or UDP, is returned.

If failure, a value of -1 is returned.

Description

The query_DPI_port function is used to obtain the port number on which the DPI capable SNMP agent at the specified host is listening for connections (TCP) or packets (UDP).

The DPI subagent programmer only needs to use this function to code the connection setup and send or await the packet. The programmer then obtains the DPI port number, finds the IP address of the agent with the lookup_host() function and then sets up a socket for communication.

This function is implicitly executed by the DPIconnect_to_agent_TCP() function, which is the function that the DPI subagent programmer would normally use. So the query_DPI_port() function is normally not used by the DPI subagent programmer.

Examples

```
#include <snmp_dpi.h>
#include /* other include files for BSD sockets and such */
int                 handle;
unsigned char      *pack_p;
long int            dpi_port;
int                 fd;
struct sockaddr_in  s,t;             /* source and target   */

dpi_port = query_DPI_port("localhost", /* get DPI port number */
                 "public",           /* for TCP, local host */
                 dpiPortForTCP);
if (dpi_port < 0) exit(1);           /* error if negative   */

host_addr = lookup_host("localhost");  /* find target IP addr */
if (host_addr == 0) exit(1);           /* unknown, that's it  */

fd = socket(AF_INET,SOCK_STREAM,0);    /* create a TCP socket */
if (fd < 0) exit(1);                   /* failure to do so    */

memset(&s,0,sizeof(s));
s.sin_family      = AF_INET;           /* set AF_INET family  */
s.sin_port        = 0;                 /* give us any port,   */
s.sin_addr.s_addr = htonl(INADDR_ANY); /* any local IPaddress */

rc = bind(fd,(struct sockaddr *)s,     /* bind our socket(fd) */
```

```
          sizeof(s_sock));            /* defined in s socket */
if (rc < 0) exit(1);                  /* failure, so exit    */

memset(&d,0,sizeof(d));
d.sin_family      = AF_INET;          /* set AF_INET family  */
d.sin_port        = htons(dpi_port);  /* set requested port  */
d.sin_addr.s_addr = host_addr;        /* destination IP addr */
                                      /* network byte order  */
rc = connect(fd,(struct sockaddr *)d, /* connect to target   */
             sizeof(d));              /* based on d sock     */
if (rc < 0) exit(1);                  /* failed, exit        */
/* now we have a socket on which to send/receive DPI packets  */
```

Related Information

The lookup_host() Function
The DPIconnect_to_agent_TCP() Function

DPI Structures

This section describes each data structure that is used in the SNMP DPI API.

Topics

The snmp_dpi_bulk_packet Structure

Structure Definition

```
struct dpi_bulk_packet {
  long int          non_repeaters;   /* count of non-repeaters*/
  long int          max_repetitions; /* max repeaters         */
  struct dpi_next_packet *varBind_p; /* ptr to varBinds, chain*/
                                     /* of dpi_next_packets   */
};
typedef struct dpi_bulk_packet       snmp_dpi_bulk_packet;
#define snmp_dpi_bulk_packet_NULL_p  ((snmp_dpi_bulk_packet *)0)
```

Note: This structure is supported only in SNMP Version 2.

Structure Members

non_repeaters
The number of varBinds in the chain of dpi_next_packet structures that are to be treated as a single GETNEXT.

max_repetitions
The maximum number of repetitions for the remaining set of varBinds in dpi_next_packet structures treated as a single GETNEXT.

varBind_p

The pointer to the first varBind in the chain of dpi_next_packet structures.

Description

The snmp_dpi_bulk_packet structure represents a parse tree for a DPI GETBULK packet.

At the subagent side, the snmp_dpi_bulk_packet structure is normally created as a result of a call to pDPIpacket(). This is the case if the DPI packet is of type SNMP_DPI_GETBULK. The snmp_dpi_hdr structure then contains a pointer to an snmp_dpi_bulk_packet structure, which in turn has a pointer to a chain of one or more snmp_dpi_next_packet structures.

The DPI subagent programmer uses this structure to find out which variables instances are to be returned in a DPI RESPONSE.

Related Information

The pDPIpacket() Function
The snmp_dpi_hdr Structure
The snmp_dpi_next_packet Structure

The snmp_dpi_close_packet Structure

Structure Definition

```
struct dpi_close_packet {
  char               reason_code;    /* reason for closing    */
};
typedef struct dpi_close_packet      snmp_dpi_close_packet;
#define snmp_dpi_close_packet_NULL_p ((snmp_dpi_close_packet*)0)
```

Structure Members

reason_code
The reason for the close.

See DPI CLOSE Reason Codes for a list of valid reason codes.

Description

The snmp_dpi_close_packet structure represents a parse tree for a DPI CLOSE packet.

The snmp_dpi_close_packet structure may be created as a result of a call to pDPIpacket(). This is the case if the DPI packet is of type SNMP_DPI_CLOSE. The snmp_dpi_hdr structure then contains a pointer to a snmp_dpi_close_packet structure.

An snmp_dpi_close_packet_structure is also created as a result of a mkDPIclose() call, but the programmer never sees the structure since mkDPIclose() immediately creates a serialized DPI packet from it and then frees the structure.

It is recommended that DPI subagent programmer uses mkDPIclose() to create a DPI CLOSE packet.

Related Information

The pDPIpacket() Function
The mkDPIclose() Function
The snmp_dpi_hdr Structure

The snmp_dpi_get_packet Structure

Structure Definition

```
struct dpi_get_packet {
```

```
  char                 *object_p;   /* ptr to OID string     */
  char                 *group_p;    /* ptr to sub-tree(group)*/
  char                 *instance_p; /* ptr to rest of OID    */
  struct dpi_get_packet *next_p;    /* ptr to next in chain  */
};
typedef struct dpi_get_packet        snmp_dpi_get_packet;
#define snmp_dpi_get_packet_NULL_p   ((snmp_dpi_get_packet *)0)
```

Structure Members

object_p
: A pointer to a NULL terminated character string that represents the full OBJECT IDENTIFIER of the variable instance that is being accessed. It basically is a concatenation of the fields *group_p* and *instance_p*. Using this field is not recommended because it is only included for DPI Version 1 compatibility and it maybe withdrawn in a later version.

group_p
: A pointer to a NULL terminated character string that represents the registered sub-tree that caused this GET request to be passed to this DPI subagent. The sub-tree must have a trailing dot.

instance_p
: A pointer to a NULL terminated character string that represents the rest which is the piece following the sub-tree part of the OBJECT IDENTIFIER of the variable instance being accessed.

  Use of the term *instance_p* here should not be confused with an OBJECT instance because this string may consist of a piece of the OBJECT IDENTIFIER plus the INSTANCE IDENTIFIER.

next_p
: A pointer to a possible next snmp_dpi_get_packet structure. If this next field contains the NULL pointer, this is the end of the chain.

Description

The snmp_dpi_get_packet structure represents a parse tree for a DPI GET packet.

At the subagent side, the snmp_dpi_get_packet structure is normally created as a result of a call to pDPIpacket(). This is the case if the DPI packet is of type SNMP_DPI_GET. The snmp_dpi_hdr structure then contains a pointer to a chain of one or more snmp_dpi_get_packet structures.

The DPI subagent programmer uses this structure to find out which variables instances are to be returned in a DPI RESPONSE.

Related Information

The pDPIpacket() Function
The snmp_dpi_hdr Structure

The snmp_dpi_hdr Structure

Structure Definition

```
struct snmp_dpi_hdr {
  unsigned char  proto_major;   /* always 2: SNMP_DPI_PROTOCOL*/
  unsigned char  proto_version; /* DPI version                */
  unsigned char  proto_release; /* DPI release                */
  unsigned short packet_id;     /* 16-bit, DPI packet ID      */
  unsigned char  packet_type;   /* DPI packet type            */
  union {
     snmp_dpi_reg_packet      *reg_p;
     snmp_dpi_ureg_packet     *ureg_p;
     snmp_dpi_get_packet      *get_p;
     snmp_dpi_next_packet     *next_p;
     snmp_dpi_next_packet     *bulk_p;
     snmp_dpi_set_packet      *set_p;
     snmp_dpi_resp_packet     *resp_p;
     snmp_dpi_trap_packet     *trap_p;
     snmp_dpi_open_packet     *open_p;
     snmp_dpi_close_packet    *close_p;
     unsigned char            *any_p;
```

```
  } data_u;
};
typedef struct snmp_dpi_hdr    snmp_dpi_hdr;
#define snmp_dpi_hdr_NULL_p    ((snmp_dpi_hdr *)0)
```

Structure Members

proto_major
The major protocol. For SNMP DPI, it is always 2.

proto_version
The DPI version.

proto_release
The DPI release.

packet_id
This field contains the packet ID of the DPI packet. When you create a response to a request, the packet ID must be the same as that of the request. This is taken care of if you use the mkDPIresponse() function.

packet_type
The type of DPI packet (parse tree) which you are dealing with.

See DPI Packet Types for a list of currently defined DPI packet types

data_u
A union of pointers to the different types of data structures that are created based on the *packet_type* field. The pointers themselves have names that are self-explanatory.

The fields *proto_major* , *proto_version* , *proto_release* , and *packet_id* are basically for DPI internal use. So the DPI programmer normally does not need to be concerned about them. If you work with an unreliable DPI "connection", such as UDP, you may want to use the packet_id field to ensure you are handling the correct packet.

Description

The snmp_dpi_hdr structure is the anchor of a DPI parse tree. At the subagent side, the snmp_dpi_hdr structure is normally created as a result of a call to pDPIpacket().

The DPI subagent programmer uses this structure to interrogate packets. Depending on the *packet_type* , the pointer to the chain of one or more packet_type specific structures that contain the actual packet data can be picked.

The storage for a DPI parse tree is always dynamically allocated. It is the responsibility of the caller to free this parse tree when it is no longer needed. You can use the fDPIparse() function to do that.

Note: Some mkDPIxxxx functions do free the parse tree that is passed to them. An example is the mkDPIpacket() function.

Related Information

The fDPIparse() Function
The pDPIpacket() Function
The snmp_dpi_close_packet Structure
The snmp_dpi_get_packet Structure
The snmp_dpi_next_packet Structure
The snmp_dpi_bulk_packet Structure
The snmp_dpi_resp_packet Structure
The snmp_dpi_set_packet Structure
The snmp_dpi_ureg_packet Structure

The snmp_dpi_next_packet Structure

Structure Definition

```
struct dpi_next_packet {
  char                  *object_p;  /* ptr to OID (string)   */
  char                  *group_p;   /* ptr to sub-tree(group)*/
```

```
  char                  *instance_p;/* ptr to rest of OID    */
  struct dpi_next_packet *next_p;    /* ptr to next in chain  */
};
typedef struct dpi_next_packet       snmp_dpi_next_packet;
#define snmp_dpi_next_packet_NULL_p  ((snmp_dpi_next_packet *)0)
```

Structure Members

object_p
: A pointer to a NULL terminated character string that represents the full OBJECT IDENTIFIER of the variable instance that is being accessed. It basically is a concatenation of the fields *group_p* and *instance_p*. Using this field is not recommended because it is only included for DPI Version 1 compatibility and it maybe withdrawn in a later version.

group_p
: A pointer to a NULL terminated character string that represents the registered sub-tree that caused this GETNEXT request to be passed to this DPI subagent. This sub-tree must have a trailing dot.

instance_p
: A pointer to a NULL terminated character string that represents the rest which is the piece following the sub-tree part of the OBJECT IDENTIFIER of the variable instance being accessed.

  Use of the term *instance_p* here should not be confused with an OBJECT instance because this string may consist of a piece of the OBJECT IDENTIFIER plus the INSTANCE IDENTIFIER.

next_p
: A pointer to a possible next snmp_dpi_get_packet structure. If this next field contains the NULL pointer, this is the end of the chain.

Description

The snmp_dpi_next_packet structure represents a parse tree for a DPI GETNEXT packet.

At the subagent side, the snmp_dpi_next_packet structure is normally created as a result of a call to pDPIpacket(). This is the case if the DPI packet is of type SNMP_DPI_GETNEXT. The snmp_dpi_hdr structure then contains a pointer to a chain of one or more snmp_dpi_next_packet structures.

The DPI subagent programmer uses this structure to find out which variables instances are to be returned in a DPI RESPONSE.

Related Information

The pDPIpacket() Function
The snmp_dpi_hdr Structure

The snmp_dpi_resp_packet Structure

Structure Definition

```
struct dpi_resp_packet {
  char                  error_code; /* like: SNMP_ERROR_xxx  */
  unsigned long int     error_index;/* 1st varBind in error  */
  #define resp_priority  error_index /* if respons to register*/
  struct dpi_set_packet *varBind_p;  /* ptr to varBind, chain */
                                     /* of dpi_set_packets    */
};
typedef struct dpi_resp_packet       snmp_dpi_resp_packet;
#define snmp_dpi_resp_packet_NULL_p  ((snmp_dpi_resp_packet *)0)
```

Structure Members

error_code
: The return code or the error code.

  See DPI RESPONSE Error Codes for a list of valid codes.

error_index

Specifies the first varBind is in error. Counting starts at 1 for the first varBind. This field should be zero (SNMP_ERROR_noError) if there is no error.

resp_priority
This is a redefinition of the *error_index* field. If the response is a response to a DPI REGISTER request and the error_code is equal to SNMP_ERROR_DPI_noError or SNMP_ERROR_DPI_higherPriorityRegistered, then this field contains the priority that was actually assigned. Otherwise, this field is set to zero for responses to a DPI REGISTER..

varBind_p
A pointer to the chain of one or more snmp_dpi_set_structures, representing varBinds of the response. This field contains a NULL pointer if there are no varBinds in the response.

Description

The snmp_dpi_resp_packet structure represents a parse tree for a DPI RESPONSE packet.

The snmp_dpi_resp_packet structure is normally created as a result of a call to pDPIpacket(). This is the case if the DPI packet is of type SNMP_DPI_RESPONSE. The snmp_dpi_hdr structure then contains a pointer to a snmp_dpi_resp_packet structure.

At the DPI subagent side, a DPI RESPONSE should only be expected at initialization and termination time when the subagent has issued a DPI OPEN, DPI REGISTER or DPI UNREGISTER request.

The DPI programmer is advised to use the mkDPIresponse() function to prepare a DPI RESPONSE packet.

Related Information

The pDPIpacket() Function
The mkDPIresponse() Function
The snmp_dpi_set_packet Structure
The snmp_dpi_hdr Structure

The snmp_dpi_set_packet Structure

Structure Definition

```
struct dpi_set_packet {
  char             *object_p;    /* ptr to Object ID (string) */
  char             *group_p;     /* ptr to sub-tree (group)   */
  char             *instance_p;  /* ptr to rest of OID        */
  unsigned char     value_type;  /* value type: SNMP_TYPE_xxx */
  unsigned short    value_len;   /* value length              */
  char             *value_p;     /* ptr to the value itself   */
  struct dpi_set_packet *next_p; /* ptr to next in chain      */
};
typedef struct dpi_set_packet       snmp_dpi_set_packet;
#define snmp_dpi_set_packet_NULL_p  ((snmp_dpi_set_packet *)0)
```

Structure Members

object_p
A pointer to a NULL terminated character string that represents the full OBJECT IDENTIFIER of the variable instance that is being accessed. It basically is a concatenation of the fields *group_p* and *instance_p* . Using this field is not recommended because it is only included for DPI Version 1 compatibility and it maybe withdrawn in a later version.

group_p
A pointer to a NULL terminated character string that represents the registered sub-tree that caused this SET, COMMIT, or UNDO request to be passed to this DPI subagent. The sub-tree must have a trailing dot.

instance_p
A pointer to a NULL terminated character string that represents the rest, which is the piece following the sub-tree part, of the OBJECT IDENTIFIER of the variable instance being accessed.

Use of the term *instance_p* here should not be confused with an OBJECT instance because this string may consist of a piece of the OBJECT IDENTIFIER plus the INSTANCE IDENTIFIER.

value_type
The type of the value.

See DPI SNMP Value Types for a list of currently defined value types.

value_len
This is an unsigned 16-bit integer that specifies the length in octets of the value pointed to by the *value* field. The length may be zero if the value if of type SNMP_TYPE_NULL.

value_p
A pointer to the actual value. This field may contain a NULL pointer if the value if of type SNMP_TYPE_NULL.

See Value Representation for information on how the data is represented for the various value types.

next_p
A pointer to a possible next snmp_dpi_set_packet structure. If this next field contains the NULL pointer, this is the end of the chain.

Description

The snmp_dpi_set_packet structure represents a parse tree for a DPI SET request.

The snmp_dpi_set_packet structure may be created as a result of a call to pDPIpacket(). This is the case if the DPI packet is of type SNMP_DPI_SET, SNMP_DPI_COMMIT or SNMP_DPI_UNDO. The snmp_dpi_hdr structure then contains a pointer to a chain of one or more snmp_dpi_set_packet structures.

This structure can also be created with a mkDPIset() call, which is typically used when preparing varBinds for a DPI RESPONSE packet.

Related Information

The pDPIpacket() Function
The mkDPIset() Function
DPI SNMP Value Types
Value Representation
The snmp_dpi_hdr Structure

The snmp_dpi_ureg_packet Structure

Structure Definition

```
struct dpi_ureg_packet {
  char                  reason_code;/* reason for unregister */
  char                 *group_p;    /* ptr to sub-tree(group)*/
  struct dpi_reg_packet *next_p;     /* ptr to next in chain  */
};
typedef struct dpi_ureg_packet       snmp_dpi_ureg_packet;
#define snmp_dpi_ureg_packet_NULL_p  ((snmp_dpi_ureg_packet *)0)
```

Structure Members

reason_code
The reason for the unregister.

See DPI UNREGISTER Reason Codes for a list of the currently defined reason codes.

group_p
A pointer to a NULL terminated character string that represents the sub-tree to be unregistered. This sub-tree must have a trailing dot.

next_p
A pointer to a possible next snmp_dpi_ureg_packet structure. If this next field contains the NULL pointer, this is the end of the chain. Currently we do not support multiple unregister requests in one DPI packet, so this field should always be zero.

Description

The snmp_dpi_ureg_packet structure represents a parse tree for a DPI UNREGISTER request.

The snmp_dpi_ureg_packet structure is normally created as a result of a call to pDPIpacket(). This is the case if the DPI packet is of type SNMP_DPI_UNREGISTER. The snmp_dpi_hdr structure then contains a pointer to a snmp_dpi_ureg_packet structure.

The DPI programmer is advised to use the mkDPIunregister() function to create a DPI UNREGISTER packet.

Related Information

The pDPIpacket() Function
The mkDPIunregister() Function
The snmp_dpi_hdr Structure

The snmp_dpi_u64 Structure

Structure Definition

```
struct snmp_dpi_u64 {           /* for unsigned 64-bit int */
  unsigned long high;           /* - high order 32 bits    */
  unsigned long low;            /* - low order  32 bits    */
};
typedef struct snmp_dpi_u64     snmp_dpi_u64;
```

Note: This structure is supported only in SNMP Version 2.

Structure Members

high
: The high order, most significant, 32 bits

low
: The low order, least significant, 32 bits

Description

The snmp_dpi_u64 structure represents an unsigned 64-bit integer as need for values with a type of SNMP_TYPE_Counter64.

The snmp_dpi_u64 structure may be created as a result of a call to pDPIpacket(). This is the case if the DPI packet is of type SNMP_DPI_SET and one of the values has a type of SNMP_TYPE_Counter64. The value_p pointer of the snmp_dpi_set_packet structure will then point to an snmp_dpi_u64 structure.

The DPI programmer must also use an snmp_dpi_u64 structure as the parameter to a mkDPIset() call if you want to create a value of type SNMP_TYPE_Counter64.

Related Information

The pDPIpacket() Function
The snmp_dpi_set_packet Structure
DPI SNMP Value Types
Value Representation

Character Set Selection

Based on the character set used on the platform where the agent and subagent are running, you will encounter one of the following three scenarios:

- Both run on an ASCII based platform.

  In reality a lot of platforms use the ASCII character set. For those platforms, just use the native character set. In that case, the native character set is ASCII.

- Both run on the same non-ASCII based platform.

  It is expected that the agent and the subagent normally run on the same machine or at least on the same platform. In that case, it is easiest to use the native character set for data that is represented as strings. If such native character set is not the ASCII character set, the agent must translate from ASCII to the native character set (and vice versa) as needed.

- One runs on ASCII based platform, the other on a non-ASCII based platform.

  If the agent and subagent each run on their own platform and those platforms use different native character sets; for example, IBM OS/2 uses ASCII and IBM MVS uses EBCDIC, you must select the ASCII character set, so that you both know exactly how to represent string-based data that is being send back and forth. The entity that is not ASCII based must do the translation from ASCII to the native character set (and vice versa) as needed.

When the DPI subagent sends a DPI OPEN packet, it must specify the character set that it wants to use. The subagent here needs to know or determine in an implementation dependent manner if the agent is running on a system with the same character set as the subagent. If you connect to the agent at loopback, localhost, or your own machine, you might assume that you are using the same character set. As long as you are just using OS/2 on an Intel based processor, it does not matter. Always use the native character set, which is ASCII.

The subagent has two choices:

DPI_NATIVE_CSET	Specifies that you want to use the native character set of the platform on which the agent that you connect to is running.
DPI_ASCII_CSET	Specifies that you want to use the ASCII character set. The agent will translate between ASCII and the native character set as required. If the subagent is on a non-ASCII based platform, it may have to translate also.

The DPI packets have a number of fields that are represented as strings. The fields that must be represented in the selected character set are:

- The null terminated string pointed to by the *description_p, enterprise_p, group_p, instance_p,* and *oid_p* parameters in the various *mkDPIxxxx(...)* functions.
- The string pointed to by the *value_p* parameter in the *mkDPIset(...)* function, that is if the *value_type* parameter specifies that the value is an *SNMP_TYPE_DisplayString* or an *SNMP_TYPE_OBJECT_IDENTIFIER*.
- The null terminated string pointed to by the *description_p, enterprise_p, group_p, instance_p,* and *oid_p* pointers in the various *snmp_dpi_xxxx_packet* structures.
- The string pointed to by the *value_p* pointer in the *snmp_dpi_set_packet* structure, that is if the *value_type* field specifies that the value is an *SNMP_TYPE_DisplayString* or an *SNMP_TYPE_OBJECT_IDENTIFIER*.

Related Information

The mkDPIopen() Function

Constants, Values, Return Codes, and Include File

This section describes all the constants and names for values as they are defined in the snmp_dpi.h include file.

Topics

DPI CLOSE Reason Codes

The currently defined DPI CLOSE reason codes as defined in the snmp_dpi.h include file are:

```
#define SNMP_CLOSE_otherReason                1
#define SNMP_CLOSE_goingDown                  2
#define SNMP_CLOSE_unsupportedVersion         3
#define SNMP_CLOSE_protocolError              4
#define SNMP_CLOSE_authenticationFailure      5
#define SNMP_CLOSE_byManager                  6
#define SNMP_CLOSE_timeout                    7
#define SNMP_CLOSE_openError                  8
```

These codes are used in the *reason_code* parameter for the *mkDPIclose()* function and in the *reason_code* field in the *snmp_dpi_close_packet* structure.

Related Information

The snmp_dpi_close_packet Structure
The mkDPIclose() Function

DPI Packet Types

The currently defined DPI packet types as defined in the snmp_dpi.h include file are:

```
#define SNMP_DPI_GET             1
#define SNMP_DPI_GET_NEXT        2  /* old DPI 1.x style */
#define SNMP_DPI_GETNEXT         2
#define SNMP_DPI_SET             3
#define SNMP_DPI_TRAP            4
#define SNMP_DPI_RESPONSE        5
#define SNMP_DPI_REGISTER        6
#define SNMP_DPI_UNREGISTER      7
#define SNMP_DPI_OPEN            8
#define SNMP_DPI_CLOSE           9
#define SNMP_DPI_COMMIT         10
#define SNMP_DPI_UNDO           11
#define SNMP_DPI_GETBULK        12
#define SNMP_DPI_TRAPV2         13  /* reserved, not ....    */
#define SNMP_DPI_INFORM         14  /* reserved, implemented */
#define SNMP_DPI_ARE_YOU_THERE  15
```

These packet types are used in the *type* parameter for the *packet_type* field in the *snmp_dpi_hdr* structure.

Related Information

The snmp_dpi_hdr Structure

DPI RESPONSE Error Codes

In case of an error on an SNMP request like GET, GETNEXT, GETBULK, SET, COMMIT, or UNDO, the RESPONSE can have one of these currently defined error codes. They are defined in the snmp_dpi.h include file:

```
#define SNMP_ERROR_noError              0
#define SNMP_ERROR_tooBig               1
#define SNMP_ERROR_noSuchName           2
#define SNMP_ERROR_badValue             3
#define SNMP_ERROR_readOnly             4
#define SNMP_ERROR_genErr               5
#define SNMP_ERROR_noAccess             6
```

```
#define SNMP_ERROR_wrongType             7
#define SNMP_ERROR_wrongLength           8
#define SNMP_ERROR_wrongEncoding         9
#define SNMP_ERROR_wrongValue           10
#define SNMP_ERROR_noCreation           11
#define SNMP_ERROR_inconsistentValue    12
#define SNMP_ERROR_resourceUnavailable  13
#define SNMP_ERROR_commitFailed         14
#define SNMP_ERROR_undoFailed           15
#define SNMP_ERROR_authorizationError   16
#define SNMP_ERROR_notWritable          17
#define SNMP_ERROR_inconsistentName     18
```

In case of an error on a DPI only request (OPEN, REGISTER, UNREGISTER, ARE_YOU_THERE), the RESPONSE can have one of these currently defined error codes. They are defined in the snmp_dpi.h include file:

```
#define SNMP_ERROR_DPI_noError                        0
#define SNMP_ERROR_DPI_otherError                   101
#define SNMP_ERROR_DPI_notFound                     102
#define SNMP_ERROR_DPI_alreadyRegistered            103
#define SNMP_ERROR_DPI_higherPriorityRegistered     104
#define SNMP_ERROR_DPI_mustOpenFirst                105
#define SNMP_ERROR_DPI_notAuthorized                106
#define SNMP_ERROR_DPI_viewSelectionNotSupported    107
#define SNMP_ERROR_DPI_getBulkSelectionNotSupported 108
#define SNMP_ERROR_DPI_duplicateSubAgentIdentifier  109
#define SNMP_ERROR_DPI_invalidDisplayString         110
#define SNMP_ERROR_DPI_characterSetSelectionNotSupported  111
```

These codes are used in the *error_code* parameter for the *mkDPIresponse()* function and in the *error_code* field in the *snmp_dpi_resp_packet* structure.

Related Information

The snmp_dpi_resp_packet Structure
The mkDPIresponse() Function

DPI UNREGISTER Reason Codes

These are the currently defined DPI UNREGISTER reason codes. They are define in the snmp_dpi.h include file:

```
#define SNMP_UNREGISTER_otherReason              1
#define SNMP_UNREGISTER_goingDown                2
#define SNMP_UNREGISTER_justUnregister           3
#define SNMP_UNREGISTER_newRegistration          4
#define SNMP_UNREGISTER_higherPriorityRegistered 5
#define SNMP_UNREGISTER_byManager                6
#define SNMP_UNREGISTER_timeout                  7
```

These codes are used in the *reason_code* parameter for the *mkDPIunregister()* function and in the *reason_code* field in the *snmp_dpi_ureg_packet* structure.

Related Information

The snmp_dpi_ureg_packet Structure
The mkDPIunregister() Function

DPI SNMP Value Types

These are the currently defined value types as defined in the snmp_dpi.h include file:

```
#define SNMP_TYPE_MASK           0x7f  /* mask to isolate type*/
#define SNMP_TYPE_Integer32    (128|1) /* 32-bit INTEGER      */
#define SNMP_TYPE_OCTET_STRING      2  /* OCTET STRING        */
#define SNMP_TYPE_OBJECT_IDENTIFIER 3  /* OBJECT IDENTIFIER   */
#define SNMP_TYPE_NULL              4  /* NULL, no value      */
#define SNMP_TYPE_IpAddress         5  /* IMPLICIT OCTETSTRING*/
#define SNMP_TYPE_Counter32    (128|6) /* 32-bit Counter      */
#define SNMP_TYPE_Gauge32      (128|7) /* 32-bit Gauge        */
#define SNMP_TYPE_TimeTicks    (128|8) /* 32-bit TimeTicks in */
                                       /* hundredths of a sec */
#define SNMP_TYPE_DisplayString     9  /* DisplayString (TC)  */
#define SNMP_TYPE_BIT_STRING       10  /* BIT STRING          */
#define SNMP_TYPE_NsapAddress      11  /* IMPLICIT OCTETSTRING*/
#define SNMP_TYPE_UInteger32  (128|12) /* 32-bit INTEGER      */
#define SNMP_TYPE_Counter64        13  /* 64-bit Counter      */
#define SNMP_TYPE_Opaque           14  /* IMPLICIT OCTETSTRING*/
#define SNMP_TYPE_noSuchObject     15  /* IMPLICIT NULL       */
#define SNMP_TYPE_noSuchInstance   16  /* IMPLICIT NULL       */
#define SNMP_TYPE_endOfMibView     17  /* IMPLICIT NULL       */
```

These value types are used in the *value_type* parameter for the *mkDPIset()* function and in the *value_type* field in the *snmp_dpi_set_packet* structure.

Related Information

The snmp_dpi_set_packet Structure
The mkDPIset() Function
Value Representation
Value Ranges and Limits

Value Representation

Values in the snmp_dpi_set_packet structure are represented as follows:

- 32-bit integers are defined as long int or unsigned long int. We assume that a long int is 4 bytes.
- 64-bit integers are represented as an snmp_dpi_u64.

  We only deal with unsigned 64 bit integers in SNMP. In a structure that has two fields, the high order piece and the low order piece, each is of type unsigned long int. We assume these are 4-bytes.
- Object Identifiers are NULL terminated strings in the selected character set, representing the OID in ASN.1 dotted notation. The length includes the terminating NULL.

  An ASCII example:

  ```
  '312e332e362e312e322e312e312e312e3000'h
  ```

  represents "1.3.6.1.2.1.1.1.0" which is sysDescr.0.

  An EBCDIC example:

  ```
  'f14bf34bf64bf14bf24bf14bf14bf14bf000'h
  ```

  represents "1.3.6.1.2.1.1.1.0" which is sysDescr.0.
- DisplayStrings are in the selected character set. The length specifies the length of the string.

  An ASCII example:

  ```
  '6162630d0a'h
  ```

  represents "abc\r\n", no NULL.

An EBCDIC example:

```
'8182830d25'h
```

represents "abc\r\n", no NULL.

- IpAddress, NsapAddress, and Opaque are implicit OCTET_STRING, so they are a sequence of octets/bytes. This means, for instance, that the IP address is in network byte order.
- NULL has a zero length for the value, no value data, so a NULL pointer in the *value_p* field.
- noSuchObject, noSuchInstance, and endOfMibView are implicit NULL and represented as such.
- BIT_STRING is an OCTET_STRING of the form uubbbb...bb, where the first octet (uu) is 0x00-0x07 and indicates the number of unused bits in the last octet (bb). The bb octets represent the bit string itself, where bit zero (0) comes first and so on.

Related Information

Value Ranges and Limits

Value Ranges and Limits

The following rules apply to object IDs in ASN.1 notation:

- The object ID consists of 1 to 128 subIDs, which are separated by dots.
- Each subID is a positive number. No negative numbers are allowed.
- The value of each number cannot exceed 4294967295 (4,294,967,295). This value is 2 to the power of 32 minus 1.
- The valid values of the first subID are: 0, 1, or 2.
- If the first subID has a value of 0 or 1, the second subID can only have a value of 0 through 39.

The following rules apply to DisplayString:

- A DisplayString (Textual Convention) is basically an OCTET STRING in SNMP terms.
- The maximum size of a DisplayString is 255 octets/bytes.
- The octets of a DisplayString must belong to the ASCII NVT character set. This character set is not precisely defined, but commonly accepted to consist of all ASCII characters with a value in the range of 0-127 inclusive.

  A further limitation is that the CR (hex 13) must always be followed by either a LF (hex 0A) or a NUL (hex 00).

More information on the DPI SNMP value types can be found in the SNMP SMI (Structure of Management Information) and SNMP TC (Textual Conventions) RFCs. At the time of this publication, these two RFCs are RFC1442 and RFC1443.

Return Codes from DPI Transport-Related Functions

These are the currently defined values for the return codes from DPI transport-related functions. They are defined in the snmp_dpi.h include file:

```
#define DPI_RC_OK                0 /* all OK, no error         */
#define DPI_RC_NOK              -1 /* some other error         */
#define DPI_RC_NO_PORT          -2 /* can't determine DPIport  */
#define DPI_RC_NO_CONNECTION    -3 /* no connection to DPIagent*/
#define DPI_RC_EOF              -4 /* EOF receivd on connection*/
#define DPI_RC_IO_ERROR         -5 /* Some I/O error on connect*/
#define DPI_RC_INVALID_HANDLE   -6 /* unknown/invalid handle   */
```

```
#define DPI_RC_TIMEOUT          -7 /* timeout occurred         */
#define DPI_RC_PACKET_TOO_LARGE -8 /* packed too large, dropped*/
```

These values are used as return codes for the transport-related DPI functions.

Related Information

The DPIconnect_to_agent_TCP() Function
The DPIconnect_to_agent_SHM() Function
The DPIawait_packet_from_agent() Function
The DPIsend_packet_to_agent() Function

The snmp_dpi.h Include File

Syntax

```
#include <snmp_dpi.h>
```

Parameters

None.

Description

The snmp_dpi.h include file defines the SNMP DPI API to the DPI subagent programmer. It has all the function proto-type statements, and it also has the definitions for the snmp_dpi structures.

The same include file is used at the agent side, so you will see some definitions which are unique to the agent side. Also there may be other functions or prototypes of functions not implemented on OS/2. Therefore, you should only use the API as far as it is documented in this information.

Related Information

Macros, functions, structures, constants and values defined in the snmp_dpi.h include file are:

The DPIawait_packet_from_agent() Function
The DPIconnect_to_agent_TCP() Function
The DPIdebug() Function
The DPIdisconnect_from_agent() Function
The DPI_PACKET_LEN() Macro
The DPIsend_packet_to_agent() Function
The fDPIparse() Function
The fDPIset() Function
The mkDPIAreYouThere() Function
The mkDPIclose() Function
The mkDPIopen() Function
The mkDPIregister() Function
The mkDPIresponse() Function
The mkDPIset() Function
The mkDPItrap() Function
The mkDPIunregister() Function
The pDPIpacket() Function
The snmp_dpi_close_packet Structure
The snmp_dpi_get_packet Structure
The snmp_dpi_next_packet Structure
The snmp_dpi_bulk_packet Structure
The snmp_dpi_hdr Structure
The lookup_host() Function
The query_DPI_port() Function
The snmp_dpi_resp_packet Structure
The snmp_dpi_set_packet Structure
The snmp_dpi_ureg_packet Structure
DPI CLOSE Reason Codes

SNMP DPI API Version 1.1 Considerations

The information presented in this section **must be taken as guidelines and not exact procedures**. Your specific implementation will vary from the guidelines presented.

You can keep your existing DPI 1.1 subagent and communicate with a DPI capable agent that supports DPI 1.1 in addition to DPI 2.0. For example, the SystemView Agent provides support for multiple versions of DPI, namely Version 1.1 and Version 2.0.

Note: Although DPI Version 1.1 is supported, it is recommended that you migrate to Version 2.0.

Normally you would compile your DPI 1.1 subagent with the DPI 1.1 <dpi\snmp_dpi.h> include file and link-edit it with the DPI 1.1 level DPI32DLL.LIB. At run time, you then need access to the DPI32DLL.DLL. You can continue to do this until you are ready to migrate to DPI Version 2.0.

In the snmp_dpi.h include file for DPI 2.0, you may find references to DPI 1.1 compatibility mode under control of compiler flags, such as:

```
/DSNMP_DPI_VERSION=1 /DSNMP_DPI_RELEASE=0
/DSNMP_DPI_VERSION=1 /DSNMP_DPI_RELEASE=1
```

However, this compatibility mode is **not** provided with the SystemView Agent. If you want to convert to DPI 2.0, which prepares you also for SNMP Version 2, you must make changes to your code.

Name Changes

A number of field names in the snmp_dpi_xxxx_packet structures have changed so that the names are now more consistent throughout the DPI code.

The new names indicate if the value is a pointer (_p) or a union (_u). The names that have changed and that affect the subagent code are listed in the table below.

OLD NAME	NEW NAME	DATA STRUCTURE(XXXX)
group_id	group_p	getnext
object_id	object_p	get, getnext, set
value	value_p	set
type	value_type	set
next	next_p	set
enterprise	enterprise_p	trap
packet_body	data_u	dpi_hdr
dpi_get	get_p	hdr (packet_body)
dpi_getnext	next_p	hdr (packet_body)
dpi_set	set_p	hdr (packet_body)
dpi_trap	trap_p	hdr (packet_body)

There is no clean approach to make this change transparent. You probably will have to change the names in your code. You may want to try a simple set of defines like:

```
#define packet_body      data_u
#define dpi_get          get_p
#define dpi_set          set_p
#define dpi_next         next_p
#define dpi_response     resp_p
#define dpi_trap         trap_p
#define group_id         group_p
#define object_id        object_p
#define value            value_p
#define type             value_type
#define next             next_p
#define enterprise       enterprise_p
```

However, the names may conflict with other definitions that you have, in which case you must change your code.

Related Information

Migrating Your SNMP DPI Subagent to Version 2.0

Migrating Your SNMP DPI Subagent to Version 2.0

The information presented in this section must be taken as guidelines and not exact procedures. Your specific implementation will vary from the guidelines presented.

When you want to change your DPI 1.x based subagent code to the DPI Version 2.0 level use these guidelines for the required actions and the recommended actions.

Required Actions

- Add a mkDPIopen() call and send the created packet to the agent. This opens your "DPI connection" with the agent. Wait for the response and ensure that the open is accepted. You need to pass a subagent ID (Object Identifier) which must be a unique ASN.1 OID.

  See The mkDPIopen() Function for more information.

- Change your mkDPIregister() calls and pass the parameters according to the new function prototype. You must also expect a RESPONSE to the REGISTER request.

  See The mkDPIregister() Function for more information.

- Change mkDPIset() and/or mkDPIlist() calls to the new mkDPIset() call. Basically all mkDPIset() calls are now of the DPI 1.1 mkDPIlist() form.

  See The mkDPIset() Function for more information.

- Change mkDPItrap() and mkDPItrape() calls to the new mkDPItrap() call. Basically all mkDPItrap() calls are now of the DPI 1.1 mkDPItrape() form.

  See The mkDPItrap() Function for more information.

- Add code to recognize DPI RESPONSE packets, which should be expected as a result of OPEN, REGISTER, UNREGISTER requests.

- Add code to expect and handle the DPI UNREGISTER packet from the agent. It may send such packets if an error occurs or if a higher priority subagent registers the same sub-tree as you have registered.

- Add code to unregister your sub-tree(s) and close the "DPI connection" when you want to terminate the subagent.

  See The mkDPIunregister() Function and The mkDPIclose() Function for more information.

- Change your code to use the new SNMP Version 2 error codes as defined in the snmp_dpi.h include file.

- Change your code that handles a GET request. It should return a varBind with SNMP_TYPE_noSuchObject value or SNMP_TYPE_noSuchInstance value instead of an error SNMP_ERROR_noSuchName if the object or the instance do not exist. This is not considered an error any more. Therefore, you should return an SNMP_ERROR_noError with an error index of zero.
- Change your code that handles a GETNEXT request. It should return a varBind with SNMP_TYPE_endOfMibView value instead of an error SNMP_ERROR_noSuchName if you reach the end of your MIB or sub-tree. This is not considered an error any more. Therefore, you should return an SNMP_ERROR_noError with an error index of zero.
- Change your code that handles SET requests to follow the two phase SET/COMMIT scheme as described in SET Processing.

  See the sample handling of SET/COMMIT/UNDO in Processing a SET/COMMIT/UNDO Request.

Recommended Actions

- Do not reference the object ID pointer (object_p) in the snmp_dpi_xxxx_packet structures anymore. Instead start using the group_p and instance_p pointers. The object_p pointer may be removed in a future version of the DPI API.
- Check Transport-Related DPI API Functions to see if you want to use those functions instead of using your own code for those functions.
- Consider using more than 1 varBind per DPI packet. You can specify this on the REGISTER request. You must then be prepared to handle multiple varBinds per DPI packet. The varBinds are chained via the various snmp_dpi_xxxx_packet structures.

  See The mkDPIregister() Function for more information.
- Consider specifying a time out when you issue a DPI OPEN or DPI REGISTER.

  See The mkDPIopen() Function and The mkDPIregister() Function for more information.

A DPI Subagent Example

This is an example of a DPI subagent. The code is in the "dpi_samp.c" file.

Note: The example code presented here was copied from the sample file at the time of the publication. For the most up-to-date example code, please see the SVA\DPI directory. There may be differences in the code presented and the code that is shipped with the product. Always use the code provided in the SVA\DPI directory as the authoritative sample code.

Topics

Related Information

Subagent Programming Concepts

Overview of Subagent Processing

This overview assumes that the subagent communicates with the agent over a TCP connection. Other connection implementations are possible and, in that case, the processing approach may be a bit different.

We also take a simplistic approach in the sense that we will request the agent to send us at most one varBind per DPI packet, so we do not need to loop through a list of varBinds. Potentially, you may gain performance improvements if you allow for multiple varBinds per DPI packet on GET, GETNEXT, SET requests, but to do so, your code will have to loop through the varBind list and so it becomes somewhat more complicated. We assume that the DPI subagent programmer can handle that once you understand the basics of the DPI API.

Here is the dpiSimple MIB definition as it is implemented by the sample code, which follows:

```
DPISimple-MIB DEFINITIONS ::= BEGIN

   IMPORTS
        MODULE-IDENTITY, OBJECT-TYPE, snmpModules, enterprises
                    FROM SNMPv2-SMI
        DisplayString
                    FROM SNMPv2-TC

   ibm      OBJECT IDENTIFIER ::= { enterprises 2 }
   ibmDPI   OBJECT IDENTIFIER ::= { ibm 2 }
   dpi20MIB OBJECT IDENTIFIER ::= { ibmDPI 1 }


   dpiSimpleMIB OBJECT IDENTIFIER ::= { dpi20MIB 5 }

   dpiSimpleInteger         OBJECT-TYPE
        SYNTAX  INTEGER
        ACCESS  read-only
        STATUS  mandatory
        DESCRIPTION
            "A sample integer32 value"
        ::= { dpiSimpleMIB 1 }

   dpiSimpleString          OBJECT-TYPE
        SYNTAX  DisplayString
        ACCESS  read-write
        STATUS  mandatory
        DESCRIPTION
            "A sample Display String"
        ::= { dpiSimpleMIB 2 }

   dpiSimpleCounter32       OBJECT-TYPE
        SYNTAX  Counter     -- Counter32 is SNMPv2
        ACCESS  read-only
        STATUS  mandatory
        DESCRIPTION
            "A sample 32-bit counter"
        ::= { dpiSimpleMIB 3 }

   dpiSimpleCounter64       OBJECT-TYPE
        SYNTAX  Counter64   -- Counter64 is SNMPv2,
                            -- Not supported by SNMPv1 agents
        ACCESS  read-only
        STATUS  mandatory
        DESCRIPTION
            "A sample 64-bit counter"
        ::= { dpiSimpleMIB 4 }
END
```

To make the code more readable, we have defined the following names in our dpi_samp.c source file.

```
#define DPI_SIMPLE_SUBAGENT   "1.3.6.1.4.1.2.2.1.5"
#define DPI_SIMPLE_MIB        "1.3.6.1.4.1.2.2.1.5."
#define DPI_SIMPLE_INTEGER    "1.0"  /* dpiSimpleInteger.0   */
#define DPI_SIMPLE_STRING     "2.0"  /* dpiSimpleString.0    */
#define DPI_SIMPLE_COUNTER32  "3.0"  /* dpiSimpleCounter32.0 */
#define DPI_SIMPLE_COUNTER64  "4.0"  /* dpiSimpleCounter64.0 */
```

In addition, we have defined the following variables as global variable in our dpi_samp.c source file.

```
static int handle;                        /* handle has global scope */
static long int      value1      = 5;
#define              value2_p      cur_val_p   /* writable object    */
#define              value2_len    cur_val_len /* writable object    */
static char         *cur_val_p   = (char *)0;
static char         *new_val_p   = (char *)0;
static char         *old_val_p   = (char *)0;
```

```
static unsigned long cur_val_len = 0;
static unsigned long new_val_len = 0;
static unsigned long old_val_len = 0;
static unsigned long value3      = 1;
static snmp_dpi_u64  value4      = {0x80000000,1L};
```

Connecting to the Agent

Before a subagent can receive or send any DPI packets from/to the SNMP DPI capable agent, it must "connect" to the agent and identify itself to the agent.

The following example code returns a response. We assume that there are no errors in the request, but proper code should do the checking for that. We do proper checking for lexicographic next object, but we do no checking for ULONG_MAX, or making sure that the instance ID is indeed valid (digits and dots). If we get to the end of our dpiSimpleMIB, we must return an endOfMibView as defined by the SNMP Version 2 rules.

- A host name or IP address in dot notation that specifies where the agent is running. Often the name "loopback" or "localhost" can be used if the subagent runs on the same system as the agent.
- A community name which is used to obtain the DPI TCP port from the agent. Internally that is done by sending a regular SNMP GET request to the agent. In an open environment, we probably can use the well know community name "public".

The function returns a negative error code if an error occurs. If the connection setup is successful, it returns a handle which represents the connection and which we must use on subsequent calls to send or await DPI packets.

The second step is to identify the subagent to the agent. This is done by making a DPI-OPEN packet, sending it to the agent, and then awaiting the response from the agent. The agent may accept or deny the OPEN request. Making a DPI-OPEN packet is done by calling mkDPIopen() which expects the following parameters:

- A unique subagent identification (an Object Identifier).
- A description which can be the NULL string ("").
- Overall subagent timeout in seconds. The agent uses this value as a timeout value for a response when it sends a request to the subagent. The agent may have a maximum value for this timeout that will be used if you exceed it.
- The maximum number of varBinds per DPI packet that the subagent is willing or is able to handle.
- The character set we want to use. In most cases you want to use the native character set.
- Length of a password. A zero means no password.
- Pointer to the password or NULL if no password. It depends on the agent if subagents must specify a password to open up a connection.

The function returns a pointer to a static buffer holding the DPI packet if successful. If it fails, it returns a NULL pointer.

Once the DPI-OPEN packet has been created, you must send it to the agent. You can use the DPIsend_packet_to_agent() function which expects the following parameters:

- The handle of a connection from DPIconnect_to_agent_TCP.
- A pointer to the DPI packet from mkDPIopen.
- The length of the packet. The snmp_dpi.h include file provides a macro DPI_PACKET_LEN that calculates the packet length of a DPI packet.

This function returns DPI_RC_OK (value zero) if successful. Otherwise, an appropriate DPI_RC_xxxx error code as defined in snmp_dpi.h is returned.

Now we must wait for a response to the DPI-OPEN. To await such a response, you call the DPIawait_packet_from_agent() function which expects the following parameters:

- The handle of a connection from DPIconnect_to_agent_TCP.
- A timeout in seconds, which is the maximum time to wait for response.

- A pointer to a pointer, which will receive a pointer to a static buffer containing the awaited DPI packet. If the system fails to receive a packet, a NULL pointer is stored.
- A pointer to a long integer (32-bit), which will receive the length of the awaited packet. If it fails, it will be set to zero.

This function returns DPI_RC_OK (value zero) if successful. Otherwise, an appropriate DPI_RC_xxxx error code as defined in snmp_dpi.h is returned.

The last step is to ensure that we received a DPI-RESPONSE back from the agent. If we did, then we must ensure that the agent accepted us as a valid subagent. This will be shown by the error_code field in the DPI response packet.

The following example code establishes a connection and "opens" it by identifying yourself to the agent.

```
#include <snmp_dpi.h>              /* DPI 2.0 API definitions */
static int handle;                 /* handle has global scope */

static void do_connect_and_open(char *hostname_p, char *community_p) {
       unsigned char *packet_p;
       int            rc;
       unsigned long  length;
       snmp_dpi_hdr  *hdr_p;
       if (shared_mem) {           /* if shared memory wanted */
          handle =                 /* then (SHM) connect to   */
             DPIconnect_to_agent_SHM(1);
                                   /* always use 1 as queueID */
       } else {
          handle =
                DPIconnect_to_agent_TCP(
                                   /* (TCP) connect to agent  */
                   hostname_p,     /* on this host            */
                   community_p);   /* snmp community name     */
       } /* endif */
       if (handle < 0) exit(1);    /* If it failed, exit      */
       packet_p = mkDPIopen(       /* Make DPI-OPEN packet    */
                    DPI_SIMPLE_SUBAGENT,
                                   /* Our identification      */
                   "Simple DPI subAgent",
                                   /* description             */
                    10L,           /* Our overall timeout     */
                    1L,            /* max varBinds/packet     */
                    DPI_NATIVE_CSET,
                                   /* native character set    */
                    0L,            /* password length         */
                    (unsigned char *)0);
                                   /* ptr to password         */



       if (!packet_p) exit(1);     /* If it failed, exit      */
       rc  = DPIsend_packet_to_agent(
                                   /* send OPEN packet        */
                handle,            /* on this connection      */
                packet_p,          /* this is the packet      */
                DPI_PACKET_LEN(packet_p));
                                   /* and this is its length  */
       if (rc != DPI_RC_OK) exit(1);
                                   /* If it failed, exit      */
       rc  = DPIawait_packet_from_agent(
                                   /* wait for response       */
                handle,            /* on this connection      */
                10,                /* timeout in seconds      */
                &packet_p,         /* receives ptr to packet  */
                &length);          /* receives packet length  */
       if (rc != DPI_RC_OK) exit(1);
                                   /* If it failed, exit      */
       hdr_p = pDPIpacket(packet_p);
                                   /* parse DPI packet        */
       if (hdr_p == snmp_dpi_hdr_NULL_p)
                                   /* If we fail to parse it  */
          exit(1);                 /* then exit               */
       if (hdr_p->packet_type != SNMP_DPI_RESPONSE) exit(1);
       rc = hdr_p->data_u.
resp_p->error_code;
       if (rc != SNMP_ERROR_DPI_noError) exit(1); }
/* end of do_connect_and_open() */
```

Registering a Sub-Tree with the Agent

After we have set up a connection to the agent and after we have identified ourselves, we must register one or more MIB sub-trees for which we want to be responsible to handle all SNMP requests.

To do so, the subagent must create a DPI-REGISTER packet and send it to the agent. The agent will then send a response to indicate success or failure of the register request.

To create a DPI-REGISTER packet, the subagent uses a call to the mkDPIregister() function, which expects these parameters:

- A timeout value in seconds for this sub-tree. If you specify zero, your overall timeout value that was specified in DPI-OPEN is used. You can specify a different value if you expect longer processing time for a specific sub-tree.
- A requested priority. Multiple subagents may register the same sub-tree at different priorities. For example, 0 is better than 1 and so on. The agent considers the subagent with the best priority to be the active subagent for the sub-tree. If you specify -1, you are asking for the best priority available. If you specify 0, you are asking for a better priority than any existing subagent may already have.
- The MIB sub-tree which you want to control. You must specify this parameter with a trailing dot.
- Your choice of GETBULK processing. You can ask the agent to map a GETBULK into multiple GETNEXT packets or to pass the GETBULK to you.

The function returns a pointer to a static buffer holding the DPI packet if successful. If it fails, it returns a NULL pointer.

Now we must send this DPI-REGISTER packet to the agent with the DPIsend_packet_to_agent() function. This is similar to sending the DPI_OPEN packet. We then wait for a response from the agent. Again, we use the DPIawait_packet_from_agent() function in the same way as we awaited a response on the DPI-OPEN request. Once we have received the response, we must check the return code to ensure that registration was successful.

The following code example demonstrates how to register one MIB sub-tree with the agent.

```
#include <snmp_dpi.h>             /* DPI 2.0 API definitions */
static int handle;                /* handle has global scope */

static void do_register(void)
{
   unsigned char *packet_p;
   int            rc;
   unsigned long  length;
   snmp_dpi_hdr  *hdr_p;

   packet_p = mkDPIregister(          /* Make DPI register    */
               3,                     /* timeout in seconds   */
               0,                     /* requested priority   */
               DPI_SIMPLE_MIB,        /* ptr to the sub-tree  */
               DPI_BULK_NO);          /* GetBulk into GetNext */

   if (!packet_p) exit(1);            /* If it failed, exit   */

   rc  = DPIsend_packet_to_agent(     /* send REGISTER packet */
            handle,                   /* on this connection   */
            packet_p,                 /* this is the packet   */
            DPI_PACKET_LEN(packet_p));/* this is its length   */

   if (rc != DPI_RC_OK) exit(1);      /* If it failed, exit   */

   rc  = DPIawait_packet_from_agent(  /* wait for response    */
            handle,                   /* on this connection   */
            3,                        /* timeout in seconds   */
            &packet_p,                /* gets ptr to packet   */
            &length);                 /* gets packet length   */

   if (rc != DPI_RC_OK) exit(1);      /* If it failed, exit   */

   hdr_p = pDPIpacket(packet_p);      /* parse DPI packet     */
   if (hdr_p == snmp_dpi_hdr_NULL_p)  /* Failed to parse it   */
      exit(1);                        /* so exit              */
```

```
   if (hdr_p->packet_type != SNMP_DPI_RESPONSE) exit(1);

   rc = hdr_p->data_u.resp_p->error_code;
   if (rc != SNMP_ERROR_DPI_noError) exit(1);
} /* end of do_register() */
```

Processing Requests from the Agent

After we have registered our sample MIB sub-tree with the agent, we must expect that SNMP requests for that sub-tree will be passed for processing by us. Since the requests will arrive in the form of DPI packets on the connection that the we have established, we go into a while loop to await DPI packets from the agent.

Since the subagent cannot know in advance which kind of packet arrives from the agent, we await a DPI packet (forever), then we parse the packet, check the packet type, and process the request based on the DPI packet type. A call to pDPIpacket, which expects as parameter a pointer to the encoded/serialized DPI packet, returns a pointer to a DPI parse tree. The pointer points to a snmp_dpi_hdr structure which looks as follows:

```
struct snmp_dpi_hdr {
  unsigned char  proto_major;
  unsigned char  proto_version;
  unsigned char  proto_release;
  unsigned short packet_id;
  unsigned char  packet_type;
  union {
     snmp_dpi_reg_packet      *reg_p;
     snmp_dpi_ureg_packet     *ureg_p;
     snmp_dpi_get_packet      *get_p;
     snmp_dpi_next_packet     *next_p;
     snmp_dpi_next_packet     *bulk_p;
     snmp_dpi_set_packet      *set_p;
     snmp_dpi_resp_packet     *resp_p;
     snmp_dpi_trap_packet     *trap_p;
     snmp_dpi_open_packet     *open_p;
     snmp_dpi_close_packet    *close_p;
     unsigned char            *any_p;
  } data_u;
};
typedef struct snmp_dpi_hdr    snmp_dpi_hdr;
#define snmp_dpi_hdr_NULL_p    ((snmp_dpi_hdr *)0)
```

With the DPI parse tree, we decide how to process the DPI packet. The following code example demonstrates the high level process of a DPI subagent.

```
#include <snmp_dpi.h>             /* DPI 2.0 API definitions */
static int handle;                /* handle has global scope */

main(int argc, char *argvÝ¨, char *envpÝ¨)
{
  unsigned char *packet_p;
  int            rc = 0;
  unsigned long  length;
  snmp_dpi_hdr  *hdr_p;

  if (argc>1) {                    /* if use passed one parm */
     if (strcmp(argvÝ1¨,"-d")==0)  /* being -d, then we      */
         DPIdebug(2);              /* turn on DPI debugging  */
  } /* endif */                    /* which shows us things  */

  do_connect_and_open();           /* connect and DPI-OPEN   */

  do_register();                   /* register our sub-tree  */

  while (rc == 0) {                /* do forever             */
   rc = DPIawait_packet_from_agent( /* wait for a DPI packet  */
           handle,                 /* on this connection     */
           -1,                     /* wait forever           */
           &packet_p,              /* receives ptr to packet */
```

```
          &length);               /* receives packet length */

   if (rc != DPI_RC_OK) exit(1);    /* If it failed, exit     */

   hdr_p = pDPIpacket(packet_p);    /* parse DPI packet       */
   if (hdr_p == snmp_dpi_hdr_NULL_p)/* If we fail to parse it */
      exit(1);                      /* then exit              */

   switch(hdr_p->packet_type) {     /* handle by DPI type     */
   case SNMP_DPI_GET:
     rc = do_get(hdr_p,
                 hdr_p->data_u.get_p);
     break;
   case SNMP_DPI_GETNEXT:
     rc = do_next(hdr_p,
                  hdr_p->data_u.next_p);
     break;




   case SNMP_DPI_SET:
   case SNMP_DPI_COMMIT:
   case SNMP_DPI_UNDO:
     rc = do_set(hdr_p,
                 hdr_p->data_u.set_p);
     break;
   case SNMP_DPI_CLOSE:
     rc = do_close(hdr_p,
                   hdr_p->data_u.close_p);
     break;
   case SNMP_DPI_UNREGISTER:
     rc = do_unreg(hdr_p,
                   hdr_p->data_u.ureg_p);
     break;
   default:
     printf("Unexpected DPI packet type %d\n",
            hdr_p->packet_type);
     rc = -1;
   } /* endswitch */
   if (rc) exit(1);
  } /* endwhile */

  return(0);
} /* end of main() */
```

Processing a GET Request

When the DPI packet is parsed, the snmp_dpi_hdr structure will show in the *packet_type* that this is a SNMP_DPI_GET packet. In that case, the *packet_body* contains a pointer to a GET-varBind, which is represented in an snmp_dpi_get_packet structure:

```
struct dpi_get_packet {
  char                  *object_p;   /* ptr to OIDstring     */
  char                  *group_p;    /* ptr to sub-tree      */
  char                  *instance_p; /* ptr to rest of OID   */
  struct dpi_get_packet  *next_p;    /* ptr to next in chain */
};
typedef struct dpi_get_packet       snmp_dpi_get_packet;
#define snmp_dpi_get_packet_NULL_p  ((snmp_dpi_get_packet *)0)
```

Assuming we have registered example sub-tree 1.3.6.1.4.1.2.2.1.5 and a GET request comes in for one variable 1.3.6.1.4.1.2.2.1.5.1.0 so that it is object 1 instance 0 in our sub-tree, the fields in the snmp_dpi_get_packet would have pointers to:

```
  object_p   ->  "1.3.6.1.4.1.2.2.1.5.1.0"
  group_p    ->  "1.3.6.1.4.1.2.2.1.5."
  instance_p ->  "1.0"
```

```
  next_p     ->  snmp_dpi_get_packet_NULL_p
```

If there are multiple varBinds in a GET request, each one is represented in a snmp_dpi_get_packet structure and all the snmp_dpi_get_packet structures are chained via the next pointer. As long as the next pointer is not the snmp_dpi_get_packet_NULL_p pointer, there are more varBinds in the list.

Now we can analyze the varBind structure for whatever checking we want to do. Once we are ready to make a response that contains the value of the variable, we prepare a SET-varBind which is represented in an snmp_dpi_set_packet structure:

```
struct dpi_set_packet {
  char                  *object_p;   /* ptr to OIDstring     */
  char                  *group_p;    /* ptr to sub-tree      */
  char                  *instance_p; /* ptr to rest of OID   */
  unsigned char          value_type; /* SNMP_TYPE_xxxx       */
  unsigned short         value_len;  /* value length         */
  char                  *value_p;    /* ptr to value itself  */
  struct dpi_set_packet  *next_p;     /* ptr to next in chain */
};
typedef struct dpi_set_packet       snmp_dpi_set_packet;
#define snmp_dpi_set_packet_NULL_p  ((snmp_dpi_set_packet *)0)
```

We can use the mkDPIset() function to prepare such a structure. This function expects the following parameters:

- A pointer to an existing snmp_dpi_set_packet structure if the new varBind must be added to an existing chain of varBinds. If this is the first or the only varBind in the chain, pass the snmp_dpi_set_packet_NULL_p pointer to indicate this.
- A pointer to the sub-tree that we registered.
- A pointer to the rest of the OID; in other words, the piece that follows the sub-tree.
- The value type of the value to be bound to the variable name. This is must be one of the SNMP_TYPE_xxxx values as defined in the snmp_dpi.h include file.
- The length of the value for integer type values. This must be a length of 4. So we always work with 32-bit signed or unsigned integers except for the Counter64 type. For the Counter64 type, we must point to a snmp_dpi_u64 structure and pass the length of that structure.
- A pointer to the value.

Memory for the varBind is dynamically allocated and the data itself is copied. So upon return we can dispose of our own pointers and allocated memory as we please. If the call is successful, a pointer is returned as follows:

- To a new snmp_dpi_set_packet if it is the first or only varBind.
- To the existing snmp_dpi_set_packet that we passed on the call. In this case, the new packed has been chained to the end of the varBind list.

If the mkDPIset() call fails, a NULL pointer is returned.

Once we have prepared the SET-varBind data, we can create a DPI RESPONSE packet using the mkDPIresponse() function which expects these parameters:

- A pointer to an snmp_dpi_hdr. We should use the header of the parsed incoming packet. It is used to copy the *packet_id* from the request into the response, such that the agent can correlate the response to a request.
- A return code which is an SNMP error code. If successful, this should be SNMP_ERROR_noError (value zero). If failure, it must be one of the SNMP_ERROR_xxxx values as defined in the snmp_dpi.h include file.

  A request for a non-existing object or instance is not considered an error. Instead, we must pass a value type of SNMP_TYPE_noSuchObject or SNMP_TYPE_noSuchInstance respectively. These two value types have an implicit value of NULL, so we can pass a zero length and a NULL pointer for the value in this case.
- The index of the varBind in error starts counting at 1. Pass zero if no error occurred, or pass the proper index of the first varBind for which an error was detected.
- A pointer to a chain of snmp_dpi_set_packets (varBinds) to be returned as response to the GET request. If an error was detected, an snmp_dpi_set_packet_NULL_p pointer may be passed.

The following code example returns a response. We assume that there are no errors in the request, but proper code should do the checking for that. For instance, we return a noSuchInstance if the instance is not exactly what we expect and a noSuchObject if the object instance_ID is greater than 3, for example 4.0. However, there might be no instance_ID at all and we should check for that too.

```
static int do_get(snmp_dpi_hdr *hdr_p,
                  snmp_dpi_get_packet *pack_p)
{
  unsigned char       *packet_p;
  int                  rc;
  snmp_dpi_set_packet *varBind_p;

  varBind_p =                         /* init the varBind chain*/
     snmp_dpi_set_packet_NULL_p;      /* to a NULL pointer     */

  if (pack_p->instance_p &&
      (strcmp(pack_p->instance_p,"1.0") == 0))
  {
    varBind_p = mkDPIset(             /* Make DPI set packet   */
                varBind_p,            /* ptr to varBind chain  */
                pack_p->group_p,      /* ptr to sub-tree       */
                pack_p->instance_p,   /* ptr to rest of OID    */
                SNMP_TYPE_Integer32, /* value type Integer 32 */
                sizeof(value1),       /* length of value       */
                &value1);             /* ptr to value          */
  } else if (pack_p->instance_p &&
      (strcmp(pack_p->instance_p,"2.0") == 0))
  {
    varBind_p = mkDPIset(                /* Make DPI set packet*/
                varBind_p,               /* ptr to varBindchain*/
                pack_p->group_p,         /* ptr to sub-tree    */
                pack_p->instance_p,      /* ptr to rest of OID */
                SNMP_TYPE_DisplayString,/* value type         */
                strlen(value2_p),        /* length of value    */
                value2_p);               /* ptr to value       */
  } else if (pack_p->instance_p &&
      (strcmp(pack_p->instance_p,"3.0") == 0))
  {
    varBind_p = mkDPIset(                /* Make DPI set packet*/
                varBind_p,               /* ptr to varBindchain*/
                pack_p->group_p,         /* ptr to sub-tree    */
                pack_p->instance_p,      /* ptr to rest of OID */
                SNMP_TYPE_Counter32,     /* value type         */
                sizeof(value3),          /* length of value    */
                &value3);                /* ptr to value       */
  } else if (pack_p->instance_p &&
      (strcmp(pack_p->instance_p,"3")>0))
  {
    varBind_p = mkDPIset(                /* Make DPI set packet*/
                varBind_p,               /* ptr to varBindchain*/
                pack_p->group_p,         /* ptr to sub-tree    */
                pack_p->instance_p,      /* ptr to rest of OID */
                SNMP_TYPE_noSuchObject,  /* value type         */
                0L,                      /* length of value    */
                (unsigned char *)0);     /* ptr to value       */
  } else {
    varBind_p = mkDPIset(                /* Make DPI set packet*/
                varBind_p,               /* ptr to varBindchain*/
                pack_p->group_p,         /* ptr to sub-tree    */
                pack_p->instance_p,      /* ptr to rest of OID */
                SNMP_TYPE_noSuchInstance,/* value type         */
                0L,                      /* length of value    */
                (unsigned char *)0);     /* ptr to value       */
  } /* endif */

  if (!varBind_p) return(-1);         /* If it failed, return  */

  packet_p = mkDPIresponse(           /* Make DPIresponse pack */
              hdr_p,                  /* ptr parsed request    */
              SNMP_ERROR_noError,     /* all is OK, no error   */
              0L,                     /* index zero, no error  */
              varBind_p);             /* varBind response data */

  if (!packet_p) return(-1);          /* If it failed, return  */

  rc  = DPIsend_packet_to_agent(      /* send RESPONSE packet  */
          handle,                     /* on this connection    */
          packet_p,                   /* this is the packet    */
          DPI_PACKET_LEN(packet_p));/* and this is its length*/

  return(rc);                         /* return retcode        */
} /* end of do_get() */
```

Processing a GETNEXT Request

When a DPI packet is parsed, the snmp_dpi_hdr structure shows in the packet_type that this is a SNMP_DPI_GETNEXT packet, and so the packet_body contains a pointer to a GETNEXT-varBind, which is represented in an snmp_dpi_next_packet structure:

```
struct dpi_next_packet {
  char                  *object_p;    /* ptr to OIDstring    */
  char                  *group_p;     /* ptr to sub-tree     */
  char                  *instance_p;  /* ptr to rest of OID  */
  struct dpi_next_packet *next_p;     /* ptr to next in chain*/
};
typedef struct dpi_next_packet      snmp_dpi_next_packet;
#define snmp_dpi_next_packet_NULL_p ((snmp_dpi_next_packet *)0)
```

In the interest of simplicity and easier understanding we will discuss the GETNEXT for a scalar object, which only has one instance. For columnar objects, which may have multiple instances, the process is more complex. However, the DPI subagent programmer should be able to handle that once the basics of GETNEXT processing in a DPI subagent is understood.

Assuming we have registered example sub-tree dpiSimpleMIB and a GETNEXT arrives for one variable, dpiSimpleInteger.0, so that is object 1 instance 0 in our sub-tree, the fields in the snmp_dpi_get_packet structure would have pointers to:

```
object_p    ->  "1.3.6.1.4.1.2.2.1.5.1.0"
group_p     ->  "1.3.6.1.4.1.2.2.1.5."
instance_p  ->  "1.0"
next_p      ->  snmp_dpi_next_packet_NULL_p
```

If there are multiple varBinds in a GETNEXT request, each one is represented in a snmp_dpi_get_packet structure and all the snmp_dpi_get_packet structures are chained via the next pointer. As long as the next pointer is not the snmp_dpi_next_packet_NULL_p pointer, there are more varBinds in the list.

Now we can analyze the varBind structure for whatever checking we want to do. We must find out which OID is the one that lexicographically follows the one in the request. It is that OID with its value that we must return as a response. Therefore, we must now also set the proper OID in the response. Once we are ready to make a response that contains the new OID and the value of that variable, we must prepare a SET-varBind which is represented in an snmp_dpi_set_packet:

```
struct dpi_set_packet {
  char                  *object_p;   /* ptr to OIDstring     */
  char                  *group_p;    /* ptr to sub-tree      */
  char                  *instance_p; /* ptr to rest of OID   */
  unsigned char          value_type; /* SNMP_TYPE_xxxx       */
  unsigned short         value_len;  /* value length         */
  char                  *value_p;    /* ptr to value itself  */
  struct dpi_set_packet  *next_p;     /* ptr to next in chain */
};
typedef struct dpi_set_packet        snmp_dpi_set_packet;
#define snmp_dpi_set_packet_NULL_p   ((snmp_dpi_set_packet *)0)
```

We can use the mkDPIset() function to prepare such a structure. This function expects the following parameters:

- A pointer to an existing snmp_dpi_set_packet structure if the new varBind must be added to an existing chain of varBinds. If this is the first or only varBind in the chain, we pass the snmp_dpi_set_packet_NULL_p pointer to indicate this.
- A pointer to the sub-tree that we registered.
- A pointer to the rest of the OID, in other words the piece that follows the sub-tree.
- The value type of the value to be bound to the variable name. This is must be one of the SNMP_TYPE_xxxx values as defined in the snmp_dpi.h include file.
- The length of the value for integer type values. This must be a length of 4. So we always work with 32-bit signed or unsigned integers except for the Counter64 type. For Counter 64 type, we must point to a snmp_dpi_u64 structure and pass the length of that structure.

- A pointer to the value.

Memory for the varBind is dynamically allocated and the data itself is copied. Upon return, we can dispose of our own pointers and allocated memory as we please. If the call is successful, a pointer is returned as follows:

- A new snmp_dpi_set_packet if it is the first or only varBind.
- The existing snmp_dpi_set_packet that we passed on the call. In this case, the new packed has been chained to the end of the varBind list.

If the mkDPIset() call fails, a NULL pointer is returned.

Once we have prepared the SET-varBind data, we can create a DPI RESPONSE packet using the mkDPIresponse() function, which expects these parameters:

- A pointer to an snmp_dpi_hdr. We should use the header of the parsed incoming packet. It is used to copy the packet_id from the request into the response, such that the agent can correlate the response to a request.
- A return code which is an SNMP error code. If successful, this should be SNMP_ERROR_noError (value zero). If failure, it must be one of the SNMP_ERROR_xxxx values as defined in the snmp_dpi.h include file.

  A request for a non-existing object or instance is not considered an error. Instead, we must pass the OID and value of the first OID that lexicographically follows the non-existing object and/or instance.

  Reaching the end of our sub-tree is not considered an error. For example, if there is no NEXT OID, this is not an error. In this situation we must return the original OID as received in the request and a value_type of SNMP_TYPE_endOfMibView. This value_type has an implicit value of NULL, so we can pass a zero length and a NULL pointer for the value.
- The index of the first varBind in error starts counting at 1. Pass zero if no error occurred, or pass the proper index of the first varBind for which an error was detected.
- A pointer to a chain of snmp_dpi_set_packet(s) (varBinds) to be returned as response to the GETNEXT request. If an error was detected, an snmp_dpi_set_packet_NULL_p pointer may be passed.

The following code example returns a response. We assume that there are no errors in the request, but proper code should do the checking for that. We do proper checking for lexicographic next object, but we do no checking for ULONG_MAX, or making sure that the instance ID is indeed valid (digits and dots). If we get to the end of our dpiSimpleMIB, we must return an endOfMibView as defined by the SNMP Version 2 rules.

```
static int do_next(snmp_dpi_hdr *hdr_p,
                   snmp_dpi_next_packet *pack_p)
{
  unsigned char       *packet_p;
  int                  rc;
  unsigned long        subid;        /* subid is unsigned     */
  unsigned long        instance;     /* same with instance    */
  char                *cp;
  snmp_dpi_set_packet *varBind_p;

  varBind_p =                        /* init the varBind chain*/
     snmp_dpi_set_packet_NULL_p;     /* to a NULL pointer     */

  if (pack_p->instance_p) {          /* we have an instance ID*/
     cp = pack_p->instance_p;        /* pick up ptr           */
     subid = strtoul(cp, &cp, 10);   /* convert subid (object)*/
     if (*cp == '.') {               /* followed by a dot ?   */
        cp++;                        /* point after it if yes */
        instance=strtoul(cp,&cp,10); /* convert real instance */
                                     /* not that we need it,we*/
        subid++;                     /* only have instance 0, */
                                     /* so NEXT is next object*/
        instance = 0;                /* and always instance 0 */
     } else {                        /* no real instance      */
        instance = 0;                /* passed, so we use 0   */
        if (subid == 0) subid++;     /* if object 0, subid 1  */
     } /* endif */
  } else {                           /* no instance ID passed */
     subid = 1;                      /* so do first object    */
     instance = 0;                   /* 0 is all we have      */
  } /* endif */

  /* we have set subid and instance such that we can basically*/
  /* process the request as a GET now. Actually, we don't even*/
  /* need instance, because all out object instances are zero.*/
```

```
  if (instance != 0) printf("Strange instance: %lu\n",instance);

  switch (subid) {
  case 1:
    varBind_p = mkDPIset(            /* Make DPI set packet   */
                varBind_p,           /* ptr to varBind chain  */
                pack_p->group_p,     /* ptr to sub-tree       */
                DPI_SIMPLE_INTEGER,  /* ptr to rest of OID    */
                SNMP_TYPE_Integer32, /* value type Integer 32 */
                sizeof(value1),      /* length of value       */
                &value1);            /* ptr to value          */
    break;
  case 2:
    varBind_p = mkDPIset(               /* Make DPI set packet*/
                varBind_p,              /* ptr to varBindchain*/
                pack_p->group_p,        /* ptr to sub-tree    */
                DPI_SIMPLE_STRING,      /* ptr to rest of OID */
                SNMP_TYPE_DisplayString,/* value type         */
                strlen(value2_p),       /* length of value    */
                value2_p);              /* ptr to value       */
    break;
  case 3:
    varBind_p = mkDPIset(               /* Make DPI set packet*/
                varBind_p,              /* ptr to varBindchain*/
                pack_p->group_p,        /* ptr to sub-tree    */
                DPI_SIMPLE_COUNTER32,   /* ptr to rest of OID */
                SNMP_TYPE_Counter32,    /* value type         */
                sizeof(value3),         /* length of value    */
                &value3);               /* ptr to value       */
    break;
  case 4:                               /*               *Apr23*/
    varBind_p = mkDPIset(               /* Make DPI set packet  */
                varBind_p,              /* ptr to varBind chain */
                pack_p->group_p,        /* ptr to sub-tree      */
                DPI_SIMPLE_COUNTER64,   /* ptr to rest of OID   */
                SNMP_TYPE_Counter64,    /* value type           */
                sizeof(value4),         /* length of value      */
                &value4);               /* ptr to value         */
    break;                              /*               *Apr23*/
  default:
    varBind_p = mkDPIset(               /* Make DPI set packet*/
                varBind_p,              /* ptr to varBindchain*/
                pack_p->group_p,        /* ptr to sub-tree    */
                pack_p->instance_p,     /* ptr to rest of OID */
                SNMP_TYPE_endOfMibView, /* value type         */
                0L,                     /* length of value    */
                (unsigned char *)0);    /* ptr to value       */
    break;
  } /* endswitch */

  if (!varBind_p) return(-1);        /* If it failed, return  */

  packet_p = mkDPIresponse(          /* Make DPIresponse pack */
               hdr_p,                /* ptr parsed request    */
               SNMP_ERROR_noError,   /* all is OK, no error   */
               0L,                   /* index zero, no error  */
               varBind_p);           /* varBind response data */

  if (!packet_p) return(-1);         /* If it failed, return  */

  rc  = DPIsend_packet_to_agent(     /* send RESPONSE packet  */
          handle,                    /* on this connection    */
          packet_p,                  /* this is the packet    */
          DPI_PACKET_LEN(packet_p));/* and this is its length*/

  return(rc);                        /* return retcode        */
} /* end of do_next() */
```

Processing a SET/COMMIT/UNDO Request

These three requests can come in one of these sequences:

- SET, COMMIT
- SET, UNDO
- SET, COMMIT, UNDO

The normal sequence is SET and then COMMIT. When we receive a SET request, we must make preparations to accept the new value. For example, check that it is for an existing object and instance, check the value type and contents to be valid, allocate memory, but we must not yet make the change.

If there are no SET errors, the next request we receive will be a COMMIT request. It is then that we must make the change, but we must also keep enough information such that we can UNDO the change later if we get a subsequent UNDO request. The latter may happen if the agent discovers any errors with other subagents while processing requests that belong to the same original SNMP SET packet. All the varBinds in the same SNMP request PDU must be processed "as if atomic".

When the DPI packet is parsed, the snmp_dpi_hdr structure shows in the *packet_type* that this is an SNMP_DPI_SET, SNMP_DPI_COMMIT, or SNMP_DPI_UNDO packet. In that case, the packet_body contains a pointer to a SET-varBind, represented in an snmp_dpi_get_packet structure. COMMIT and UNDO have same varBind data as SET upon which they follow:

```
struct dpi_set_packet {
  char                   *object_p;   /* ptr to OIDstring     */
  char                   *group_p;    /* ptr to sub-tree      */
  char                   *instance_p; /* ptr to rest of OID   */
  unsigned char           value_type; /* SNMP_TYPE_xxxx       */
  unsigned short          value_len;  /* value length         */
  char                   *value_p;    /* ptr to value itself  */
  struct dpi_set_packet  *next_p;     /* ptr to next in chain */
};
typedef struct dpi_set_packet        snmp_dpi_set_packet;
#define snmp_dpi_set_packet_NULL_p   ((snmp_dpi_set_packet *)0)
```

Assuming we have registered example sub-tree dpiSimpleMIB and a GET request comes in for one variable dpiSimpleString.0 so that is object 1 instance 0 in our sub-tree, and also assuming that the agent knows about our compiled dpiSimpleMIB so that it knows this is a DisplayString as opposed to just an arbitrary OCTET_STRING, the pointers in the snmp_dpi_set_packet structure would have pointers and values like:

```
object_p    ->  "1.3.6.1.4.1.2.2.1.5.2.0"
group_p     ->  "1.3.6.1.4.1.2.2.1.5."
instance_p  ->  "2.0"
value_type  ->  SNMP_TYPE_DisplayString
value_len   ->  8
value_p     ->  pointer to the value to be set
next_p      ->  snmp_dpi_get_packet_NULL_p
```

If there are multiple varBinds in a SET request, each one is represented in a snmp_dpi_set_packet structure and all the snmp_dpi_set_packet structures are chained via the next pointer. As long as the next pointer is not the snmp_dpi_set_packet_NULL_p pointer, there are more varBinds in the list.

Now we can analyze the varBind structure for whatever checking we want to do. Once we are ready to make a response that contains the value of the variable, we may prepare a new SET-varBind. However, by definition, the response to a successful SET is exactly the same as the SET request. So there is no need to return any varBinds. A response with SNMP_ERROR_noError and an index of zero will do. If there is an error, a response with the SNMP_ERROR_xxxx error code and an index pointing to the varBind in error (counting starts at 1) will do.

The following code example returns a response. We assume that there are no errors in the request, but proper code should do the checking for that. We also do not check if the varBind in the COMMIT and/or UNDO is the same as that in the SET request. A proper agent would make sure that is the case, but a proper subagent may want to verify that for itself. We only do one check that this is dpiSimpleString.0, and if it is not, we return a noCreation. This may not be correct, the mainline does not even return a response.

```
static int do_set(snmp_dpi_hdr *hdr_p, snmp_dpi_set_packet *pack_p)
{
       unsigned char       *packet_p;
       int                  rc;
       int                  index       = 0;
       int                  error       = SNMP_ERROR_noError;
       snmp_dpi_set_packet *varBind_p;

       varBind_p =                       /* init the varBind chain  */
          snmp_dpi_set_packet_NULL_p;    /* to a NULL pointer       */

       if (!pack_p->instance_p ||
           (strcmp(pack_p->instance_p,"2.0") != 0))
       {

          if (pack_p->instance_p &&
```

```
      (strncmp(pack_p->instance_p,"1.",2) == 0))
   {
      error = SNMP_ERROR_notWritable;
   } else if (pack_p->instance_p &&
      (strncmp(pack_p->instance_p,"2.",2) == 0))
   {
      error = SNMP_ERROR_noCreation;
   } else if (pack_p->instance_p &&
      (strncmp(pack_p->instance_p,"3.",2) == 0))
   {
      error = SNMP_ERROR_notWritable;
   } else {
      error = SNMP_ERROR_noCreation;
   } /* endif */

   packet_p = mkDPIresponse(       /* Make DPIresponse packet */
             hdr_p,                /* ptr parsed request      */
             error,                /* all is OK, no error     */
             1,                    /* index is 1, 1st varBind */
             varBind_p);           /* varBind response data   */

   if (!packet_p) return(-1);      /* If it failed, return    */

   rc  = DPIsend_packet_to_agent(  /* send RESPONSE packet    */
         handle,                   /* on this connection      */
         packet_p,                 /* this is the packet      */
         DPI_PACKET_LEN(packet_p));/* and this is its length  */

   return(rc);                     /* return retcode          */
}

switch (hdr_p->packet_type) {
case SNMP_DPI_SET:
  if ((pack_p->value_type != SNMP_TYPE_DisplayString) &&
      (pack_p->value_type != SNMP_TYPE_OCTET_STRING))
  {  /* check octet string in case agent has no compiled MIB  */
     error = SNMP_ERROR_wrongType;
     break;                        /* from switch             */
  } /* endif */
  if (new_val_p) free(new_val_p);  /* free these memory areas */
  if (old_val_p) free(old_val_p);  /* if we allocated any     */
  new_val_p   = (char *)0;
  old_val_p   = (char *)0;
  new_val_len = 0;
  old_val_len = 0;

  new_val_p =                      /* allocate memory for     */
      malloc(pack_p->value_len);   /* new value to set        */
  if (new_val_p) {                 /* If success, then also   */
     memcpy(new_val_p,             /* copy new value to our   */
            pack_p->value_p,       /* own and newly allocated */
            pack_p->value_len);    /* memory area.            */
     new_val_len = pack_p->value_len;
  } else {                         /* Else failed to malloc,  */
     error = SNMP_ERROR_genErr;    /* so that is a genErr     */
     index = 1;                    /* at first varBind        */
  } /* endif */
  break;
case SNMP_DPI_COMMIT:
  old_val_p = cur_val_p;           /* save old value for undo */
  cur_val_p = new_val_p;           /* make new value current  */
  new_val_p = (char *)0;           /* keep only 1 ptr around  */
  old_val_len = cur_val_len;       /* and keep lengths correct*/
  cur_val_len = new_val_len;
  new_val_len = 0;
  /* may need to convert from ASCII to native if OCTET_STRING */
  break;
case SNMP_DPI_UNDO:
  if (new_val_p) {                 /* free allocated memory   */
     free(new_val_p);
     new_val_p   = (char *)0;
     new_val_len = 0;
  } /* endif */
  if (old_val_p) {
     if (cur_val_p) free(cur_val_p);
     cur_val_p   = old_val_p;      /* reset to old value      */
     cur_val_len = old_val_len;
     old_val_p   = (char *)0;
     old_val_len = 0;
```

```
        } /* endif */
        break;
      } /* endswitch */

      packet_p = mkDPIresponse(          /* Make DPIresponse packet */
                  hdr_p,                 /* ptr parsed request      */
                  error,                 /* all is OK, no error     */
                  index,                 /* index is zero, no error */
                  varBind_p);            /* varBind response data   */

      if (!packet_p) return(-1);         /* If it failed, return    */

      rc  = DPIsend_packet_to_agent(     /* send RESPONSE packet    */
               handle,                   /* on this connection      */
               packet_p,                 /* this is the packet      */
               DPI_PACKET_LEN(packet_p));/* and this is its length  */

      return(rc);                        /* return retcode          */
} /* end of do_set() */
```

Processing an UNREGISTER Request

An agent can send an UNREGISTER packet if some other subagent does a register for the same sub-tree at a higher priority. An agent can also send an UNREGISTER if, for example, an SNMP manager tells it to "invalidate" the subagent connection or the registered sub-tree.

Here is an example of how to handle such a packet.

```
#include <snmp_dpi.h>          /* DPI 2.0 API definitions */

static int do_unreg(snmp_dpi_hdr *hdr_p,
                    snmp_dpi_ureg_packet *pack_p)
{
  printf("DPI UNREGISTER received from agent, reason=%d\n",
          pack_p->reason_code);
  printf("    sub-tree=%s\n",pack_p->group_p);
  DPIdisconnect_from_agent(handle);
  return(-1); /* causes exit in main loop */
} /* end of do_unreg() */
```

Processing a CLOSE Request

An agent can send a CLOSE packet if it encounters an error or for some other reason. It can also do so if an SNMP MANAGER tells it to "invalidate" the subagent connection.

Here is an example of how to handle such a packet.

```
#include <snmp_dpi.h>           /* DPI 2.0 API definitions */

static int do_close(snmp_dpi_hdr *hdr_p,
                    snmp_dpi_close_packet *pack_p)
{
  printf("DPI CLOSE received from agent, reason=%d\n",
         pack_p->reason_code);

  DPIdisconnect_from_agent(handle);
  return(-1); /* causes exit in main loop */
} /* end of do_close() */
```

Generating a TRAP

A trap can be issued at any time after a DPI OPEN was successful. To do so, you must create a trap packet and send it to the agent. With the TRAP, you can pass all sorts of varBinds if you want. In this example, we pass two varBinds one with integer data and one with an octet string. You can also pass an Enterprise ID, but with DPI 2.0, the agent will use your subagent ID as the enterprise ID if you do not pass one with the trap. In most cases that will probably be fine.

We must first prepare a varBind list chain that contains the two variables that we want to pass along with the trap. To do so we must prepare a chain of two snmp_dpi_set_packet structures, which looks like:

```
struct dpi_set_packet {
  char                   *object_p;   /* ptr to OIDstring     */
  char                   *group_p;    /* ptr to sub-tree      */
  char                   *instance_p; /* ptr to rest of OID   */
  unsigned char           value_type; /* SNMP_TYPE_xxxx       */
  unsigned short          value_len;  /* value length         */
  char                   *value_p;    /* ptr to value itself  */
  struct dpi_set_packet  *next_p;     /* ptr to next in chain */
};
typedef struct dpi_set_packet        snmp_dpi_set_packet;
#define snmp_dpi_set_packet_NULL_p   ((snmp_dpi_set_packet *)0)
```

We can use the mkDPIset() function to prepare such a structure. This function expects the following parameters:

- A pointer to an existing snmp_dpi_set_packet structure if the new varBind must be added to an existing chain of varBinds. If this is the first or the only varBind in the chain, pass the snmp_dpi_set_packet_NULL_p pointer to indicate this.
- A pointer to the sub-tree that we registered.
- A pointer to the rest of the OID, in other words, the piece that follows the sub-tree.
- The value type of the value to be bound to the variable name. This is must be one of the SNMP_TYPE_xxxx values as defined in the snmp_dpi.h include file.
- The length of the value. For integer type values, this must be a length of 4. We always work with 32-bit signed or unsigned integers except for the Counter64 type. For the Counter64 type, we must point to a snmp_dpi_u64 structure and pass the length of that structure.
- A pointer to the value.

Memory for the varBind is dynamically allocated and the data itself is copied. Upon return, we can dispose of our own pointers and allocated memory as we please. If the call is successful, a pointer is returned as follows:

- To a new snmp_dpi_set_packet if it is the first or only varBind.
- To the existing snmp_dpi_set_packet that we passed on the call. In this case, the new packed has been chained to the end of the varBind list.

If the mkDPIset() call fails, a NULL pointer is returned.

Once we have prepared the SET-varBind data, we can create a DPI TRAP packet. To do so we can use the mkDPItrap() function which expects these parameters:

- The generic trap code. Use 6 for enterprise specific trap type.
- The specific trap type. This is a type that is defined by the MIB which we are implementing. In our example we just use a 1.
- A pointer to a chain of varBinds or the NULL pointer if no varBinds need to be passed with the trap.
- A pointer to the enterprise OID if we want to use a different enterprise ID than the OID we used to identify ourselves as a subagent at DPI-OPEN time.

The following code creates an enterprise specific trap with specific type 1 and passes two varBinds. The first varBind with our object 1, instance 0, Integer32 value; the second varBind with our object 2, instance 0, Octet String. We pass no enterprise ID.

```
static int do_trap(void)
{
  unsigned char       *packet_p;
```

```
  int                 rc;
  snmp_dpi_set_packet *varBind_p;

  varBind_p =                       /* init the varBindchain */
     snmp_dpi_set_packet_NULL_p,    /* to a NULL pointer     */

  varBind_p = mkDPIset(             /* Make DPI set packet   */
               varBind_p,           /* ptr to varBind chain  */
               DPI_SIMPLE_MIB,      /* ptr to sub-tree       */
               DPI_SIMPLE_INTEGER,  /* ptr to rest of OID    */
               SNMP_TYPE_Integer32, /* value type Integer 32 */
               sizeof(value1),      /* length of value       */
               &value1);            /* ptr to value          */

  if (!varBind_p) return(-1);       /* If it failed, return  */

  varBind_p = mkDPIset(                /* Make DPI set packet*/
               varBind_p,              /* ptr to varBindchain*/
               DPI_SIMPLE_MIB,         /* ptr to sub-tree    */
               DPI_SIMPLE_STRING,      /* ptr to rest of OID */
               SNMP_TYPE_DisplayString,/* value type         */
               strlen(value2_p),       /* length of value    */
               value2_p);              /* ptr to value       */

  if (!varBind_p) return(-1);       /* If it failed, return  */



  varBind_p = mkDPIset(                /* Make DPI set packet*/
               varBind_p,              /* ptr to varBindchain*/
               DPI_SIMPLE_MIB,         /* ptr to sub-tree    */
               DPI_SIMPLE_COUNTER32,   /* ptr to rest of OID */
               SNMP_TYPE_Counter32,    /* value type         */
               sizeof(value3),         /* length of value    */
               &value3);               /* ptr to value       */

  if (!varBind_p) return(-1);       /* If it failed, return  */

  packet_p = mkDPItrap(             /* Make DPItrap packet   */
              6,                    /* enterpriseSpecific    */
              1,                    /* specific type = 1     */
              varBind_p,            /* varBind data, and use */
              (char *)0);           /* default enterpriseID  */

  if (!packet_p) return(-1);        /* If it failed, return  */

  rc  = DPIsend_packet_to_agent(    /* send TRAP packet      */
          handle,                   /* on this connection    */
          packet_p,                 /* this is the packet    */
          DPI_PACKET_LEN(packet_p));/* and this is its length*/

  return(rc);                       /* return retcode        */
} /* end of do_trap() */
```

Notices

Second Edition (September 1996)

The following paragraph does not apply to the United Kingdom or any country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This publication could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time.

It is possible that this publication may contain reference to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country.

Requests for technical information about IBM products should be made to your IBM reseller or IBM marketing representative.

Copyright Notices

Disclaimers

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program or service is not intended to state or imply that only that IBM product, program, or service may be used. Subject to IBM's valid intellectual property or other legally protectable rights, any functionally equivalent product, program, or service may be used instead of the IBM product, program, or service. The evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the responsibility of the user.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
500 Columbus Avenue
Thornwood, NY 10594
U.S.A.

Asia-Pacific users can inquire, in writing, to the IBM Director of Intellectual Property and Licensing, IBM World Trade Asia Corporation, 2-31 Roppongi 3-chome, Minato-ku, Tokyo 106, Japan.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Corporation, Department LZKS, 11400 Burnet Road, Austin, TX 78758 U.S.A. Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

Common User Access	DB2/2
DATABASE 2	FFST/2
First Failure Support Technology/2	IBM
Operating System/2	OS/2

SystemView

The following terms are trademarks of other companies:

Term	Trademark of
DMI	Desktop Management Task Force
DMTF	Desktop Management Task Force
Windows NT	Microsoft Corporation
Win32	Microsoft Corporation
X-Windows	Massachusetts Institute of Technology

Microsoft, Windows and the Windows 95 Logo are trademarks of Microsoft Corporation.

Other company, product, and service names, which may be denoted by a double asterisk (**), may be trademarks or service marks of others.

Glossary

This glossary includes terms and definitions from:

- The *American National Standard Dictionary for Information Systems* , ANSI X3.172-1990, copyright 1990 by the American National Standards Institute (ANSI). Copies may be purchased from the American National Standards Institute, 11 West 42nd Street, New York, New York 10036. Definitions are identified by the symbol (A) after the definition.
- The ANSI/EIA Standard-440-A, *Fiber Optic Terminology* . Copies may be purchased from the Electronic Industries Association, 2001 Pennsylvania Avenue, N.W., Washington, DC 20006. Definitions are identified by the symbol (E) after the definition.
- The *Information Technology Vocabulary* , developed by Subcommittee 1, Joint Technical Committee 1, of the International Organization for Standardization and the International Electrotechnical Commission (ISO/IEC JTC1/SC1). Definitions of published parts of this vocabulary are identified by the symbol (I) after the definition; definitions taken from draft international standards, committee drafts, and working papers being developed by ISO/IEC JTC1/SC1 are identified by the symbol (T) after the definition, indicating that final agreement has not yet been reached among the participating National Bodies of SC1.
- The *IBM Dictionary of Computing* , New York: McGraw-Hill, 1994.
- Internet Request for Comments: 1208, *Glossary of Networking Terms* .
- Internet Request for Comments: 1392, *Internet Users' Glossary* .
- The *Object-Oriented Interface Design: IBM Common User Access Guidelines* , Carmel, Indiana: Que, 1992.
- The *Desktop Management Interface Reference* , Revision 1.0
- The *Desktop Management Interface Specification* , Version 2.0

The following cross-references are used in this glossary:

Contrast with:	This refers to a term that has an opposed or substantively different meaning.
Synonym for:	This indicates that the term has the same meaning as a preferred term, which is defined in its proper place in the glossary.
Synonymous with:	This is a backward reference from a defined term to all other terms that have the same meaning.
See:	This refers the reader to multiple-word terms that have the same last word.
See also:	This refers the reader to terms that have a related, but not synonymous, meaning.
Deprecated term for:	This indicates that the term should not be used. It refers to a preferred term, which is defined in its proper place in the glossary.

A

---

agent

agent

1. In systems management, a user that, for a particular interaction, has assumed an agent role.
2. An entity that represents one or more managed objects by (a) emitting notifications regarding the objects and (b) handling requests from managers for management operations to modify or query the objects.
3. A system that assumes an agent role.

---

attribute

attribute

1. A characteristic that identifies and describes a managed object. The characteristic can be determined, and possibly changed, through operations on the managed object.
2. Information within a managed object that is visible at the object boundary. An attribute has a type, which indicates the range of information given by the attribute, and a value, which is within that range.
3. Variable data that is logically a part of an object and that represents a property of the object. For example, a serial number is an attribute of an equipment object.
4. In the Desktop Management Interface (DMI), a piece of information about a component.

C

---

component

component

1. Hardware or software that is part of a functional unit.
2. A part of a structured type or value, such as an array element or a record field.
3. In the Desktop Management Interface (DMI), any hardware, software, or firmware element contained in (or primarily attached to) a computer system.

---

component instrumentation

component instrumentation

In the Desktop Management Interface (DMI), the executable code that provides DMI management functionality for a particular component.

CONFIG.SYS file

CONFIG.SYS file

A file that contains configuration options for an OS/2 program installed on a workstation.

D

Desktop Management Interface (DMI)

Desktop Management Interface (DMI)

A protocol-independent set of application programming interfaces (APIs) defined by the Desktop Management Task Force (DMTF). These interfaces give management applications standardized access to information about hardware and software in a system.

Desktop Management Task Force (DMTF)

Desktop Management Task Force (DMTF)

An alliance of computer vendors which was convened to streamline the management of diverse operating systems commonly found in an enterprise. The DMTF includes industry-wide workgroups, which are actively identifying the pieces of information which are necessary to manage specific categories of devices.

DMI

DMI

Desktop Management Interface.

DMTF

DMTF

Desktop Management Task Force.

DPI

DPI

Distributed Protocol Interface.

---

discovery

discovery

The automatic detection of network topology change, for example, new and deleted nodes or interfaces.

Distributed Protocol Interface (DPI)

Distributed Protocol Interface (DPI)

An interface between a Simple Network Management Protocol (SNMP) agent and its subagents that is defined in Request for Comments (RFC) 1592.

---

drive

drive

The device used to read and write data on disks or diskettes.

E

---

event

event

1. An occurrence of significance to a task; for example, an SNMP trap, the opening of a window or a submap, or the completion of an asynchronous operation.
2. In the NetView and NETCENTER programs, a record indicating irregularities of operation in physical elements of a network.
3. In the Desktop Management Interface (DMI), a type of indication (unsolicited report) that originates from a component instrumentation. See also *indication* .

G

---

group

group

1. In the NetView/PC program, to identify a set of application programs that are to run concurrently.
2. In the Desktop Management Interface (DMI), a collection of attributes. A group with multiple instances is called a table.

H

---

host name

host name

A unique name, set at the management protocol level, that identifies a node.

I

ID

ID

Identification; identifier.

---

indication

indication

In the Desktop Management Interface (DMI), an unsolicited report, either from a component instrumentation to the service provider, or from the service provider to a management application.

---

instrumentation

instrumentation

See *component instrumentation* .

K

---

key

key

In the Desktop Management Interface (DMI), an identifier of a particular instance (row) of a table.

L

---

load

load

To move data or programs into memory.

M

---

management application

management application

In the Desktop Management Interface (DMI), code that uses the management interface (MI) to request management activity from components.

Management Information Base (MIB)

Management Information Base (MIB)

1. A collection of objects that can be accessed by means of a network management protocol.
2. A definition for management information that specifies the information available from a host or gateway and the operations allowed.
3. In OSI, the conceptual repository of management information within an open system.

Management Information Format (MIF)

Management Information Format (MIF)

In the Desktop Management Interface (DMI), the format used for describing components.

Management Interface (MI)

Management Interface (MI)

In the Desktop Management Interface (DMI), the DMI layer between management applications and the service provider.

MI

MI

Management interface.

MIB

MIB

Management Information Base.

MIF

MIF

Management Information Format.

MIF database

MIF database

In the Desktop Management Interface (DMI), the collection of known MIF files, stored by the service provider (in an implementation-specific format) for fast access.

MIF file

MIF file

In the Desktop Management Interface (DMI), a file that uses the MIF to describe a component.

O

---

object identifier (OID)

object identifier (OID)

An administratively assigned data value of the type defined in abstract syntax notation 1 (ASN.1).

OID

OID

Object ID.

P

---

process

process

1. A unique, finite course of events defined by its purpose or by its effect, achieved under defined conditions.
2. Any operation or combination of operations on data.
3. A function being performed or waiting to be performed.
4. A program in operation; for example, a daemon is a system process that is always running on the system.
5. An address space, one or more threads of control that run within that address space, and the required system resources.
6. A collection of system resources that include one or more threads of execution that perform a task.

---

protocol

protocol

1. A set of semantic and syntactic rules that determines the behavior of functional units in achieving communication. (I) Protocols can determine low-level details of machine-to-machine interfaces, such as the order in which bits from a byte are sent; they can also determine high-level exchanges between application programs, such as file transfer.
2. A set of rules governing the operation of functional units of a communication system that must be followed if communication is to take place.
3. In Open Systems Interconnection architecture, a set of semantic and syntactic rules that determine the behavior of entities in the same layer in performing communication functions. (T)

R

Request for Comments (RFC)

Request for Comments (RFC)

In Internet communications, the document series that describes a part of the Internet suite of protocols and related experiments. All Internet standards are documented as RFCs.

RFC

RFC

Request for Comments.

S

---

service provider

service provider

In the Desktop Management Interface (DMI), the code between the management interface and the component interface that arbitrates access to component instrumentation and manages the MIF database.

Simple Network Management Protocol (SNMP)

Simple Network Management Protocol (SNMP)

In the Internet suite of protocols, a network management protocol that is used to monitor routers and attached networks. SNMP is an application layer protocol. Information on devices managed is defined and stored in the application's Management Information Base (MIB).

SNMP

SNMP

Simple Network Management Protocol.

---

system

system

1. A computer and its associated devices and programs.
2. An end point of a communications link or a junction common to two or more links in a network. Systems can be processors, communication controllers, cluster controllers, terminals, workstations, clients, requesters, or servers.

T

---

table

table

In the Desktop Management Interface (DMI), a multidimensional group; a group with more than one instance.

TCP/IP

TCP/IP

Transmission Control Protocol/Internet Protocol.

---

topology

topology

The schematic arrangement of the links and nodes of a network.

Transmission Control Protocol/Internet Protocol (TCP/IP)

Transmission Control Protocol/Internet Protocol (TCP/IP)

A set of communication protocols that supports peer-to-peer connectivity functions for both local and wide-area networks.

---

trap

trap

In the Simple Network Management Protocol (SNMP), a message sent by a managed node (agent function) to a management station to report an exception condition.

U

UDP

UDP

User Datagram Protocol.

User Datagram Protocol (UDP)

User Datagram Protocol (UDP)

In the Internet suite of protocols, a protocol that provides unreliable, connectionless datagram service. It enables an application program on one machine or process to send a datagram to an application program on another machine or process. UDP uses the Internet Protocol (IP) to deliver datagrams.

Bibliography

This bibliography contains a list of publications pertaining to the product and related subjects.

SystemView Agent Publications

The following paragraphs briefly describe the library for Version 1 of the SystemView Agent program:

SystemView Agent User's Guide (SVAG-USR2)

This book provides instructions for installing the SystemView Agent package and using the MIF browser and describes communication with SNMP management applications.

SystemView Agent DMI Programmer's Guide (SVAG-DMIP)

This books describes the operation of the DMI, including the command blocks and the conventions of the MIF. The book also discusses how to enable components for DMI access and how DMI information is made available to SNMP management applications.

SystemView Agent DPI Programmer's Guide (SVAG-DPIP)

This book describes the distributed protocol interface (DPI), including programming concepts, basic API functions, and examples.

Desktop Management Task Force Publications

The following publications are available from the Desktop Management Task Force:

Desktop Management Interface Specification, Version 1.1

Internet Request for Comments Documents

The Internet protocol suite is still evolving through Requests for Comments (RFCs). New protocols are being designed and implemented by researchers and are brought to the attention of the Internet community in the form of RFCs.

As networks have grown in size and complexity, SNMP has emerged as the Internet network management standard. The following RFCs provide information relevant to SNMP and related to SystemView Agent:

RFC 1155: Structure and Identification of Management Information for TCP/IP-based Internets

This RFC provides a set of rules used to define and identify MIB objects.

RFC 1157: A Simple Network Management Protocol (SNMP)

This RFC defines the protocol used to manage objects defined in a MIB.

RFC 1212: Concise MIB Definitions

RFC 1213: Management Information Base for Network Management of TCP/IP-based Internets: MIB-II

This RFC defines a base set of managed objects to be provided in a MIB.

RFC 1592, SNMP Distributed Protocol Interface (DPI), Version 2.0

This RFC describes the protocol to allow an SNMP agent to communicate with a subagent. This allows users to dynamically add, delete, or replace MIB objects without recompiling the SNMP agent.

RFC 1901, Introduction to Community-based SNMPv2

RFC 1902, Structure of Management Information for Version 2 of the Simple Network Management Protocol (SNMPv2)

RFC 1903, Textual Conventions for Version 2 of the Simple Network Management Protocol (SNMPv2)

RFC 1904, Conformance Statements for Version 2 of the Simple Network Management Protocol (SNMPv2)

RFC 1905, Protocol Operations for Version 2 of the Simple Network Management Protocol (SNMPv2)

RFC 1906, Transport Mappings for Version 2 of the Simple Network Management Protocol (SNMPv2)

RFC 1907, Management Information Base for Version 2 of the Simple Network Management Protocol (SNMPv2)

RFC 1908, Coexistence between Version 1 and Version 2 of the Internet-standard Network Management Framework

RFC 1909, An Administrative Infrastructure for SNMPv2

RFC 1910, User-based Security Model for SNMPv2
